

Assessment of Capabilities of Extremely Low Probability of Rupture (xLPR) Software—GoldSim and SIAM Version 1.0

Prepared for

**U.S. Nuclear Regulatory Commission
Office of Nuclear Regulatory Research**

Prepared by

**Osvaldo Pensado¹
E. Lynn Tipton¹
Sitakanta Mohanty¹
Thomas Wilt¹
Robert Brient¹
Graham Chell²
George Adams¹
Kaushik Das¹
Debashis Basu¹**

**¹Center for Nuclear Waste Regulatory Analyses
San Antonio, Texas**

**²Southwest Research Institute[®]
San Antonio, Texas**

May 2011

ABSTRACT

The U.S. Nuclear Regulation Commission (NRC), Office of Nuclear Regulatory Research has entered into a cooperative program with the Electric Power Research Institute (EPRI) with the goal to develop a modular-based, probabilistic fracture mechanics tool or code capable of determining the probability of failure for reactor coolant system components. This code is referred to as extremely low probability of rupture (xLPR). The intention of the code is to encompass a range of physical processes, to be flexible to permit analysis of a variety of service conditions, and to be adaptable to accommodate evolving and improving knowledge. A pilot study was sponsored by the NRC and EPRI as a proof of concept to test (i) the feasibility of developing probabilistic models for failure mechanisms for a component of the piping coolant system, (ii) configuration management and quality assurance for multiple-party code development, and (iii) existing tools for the prompt deployment of probabilistic models (NRC and EPRI, 2011). The problem of initiation and growth of primary water stress corrosion cracks in a dissimilar metal pressurizer surge nozzle weld of the piping coolant system in a nuclear power plant was selected as the subject of the pilot study. To test existing tools for quick deployment of probabilistic models, the commercial GoldSim[®] (GoldSim Technology Group LLC, 2011) software and open source tools (SIAM-PFM) were identified to support a modular development of the xLPR code. In this report, the Center for Nuclear Waste Regulatory Analyses (CNWRA[®]) provides an independent assessment of the GoldSim and SIAM frames to support development of the xLPR code. The evaluation focused on (i) ease of use from a user perspective, (ii) readability from a model developer perspective, (iii) flexibility and adaptability from a model developer perspective, and (iv) potential of the frame for expansion of xLPR. Two of the appendices report also address several other aspects related to the framework development, including elements of a robust quality assurance program; the configuration management system implemented to support controlled development of the xLPR; tasks to develop model validation, in the context of a quality assurance program; and results of a limited code verification of FORTRAN modules common to the xLPR-GoldSim and xLPR-SIAM codes. In the independent CNWRA analysis, xLPR-GoldSim and xLPR-SIAM were found to have different limitations and strengths in regard to future development potential. In general, for xLPR-GoldSim, the GoldSim environment offers convenience at the cost of workarounds and the need for external tools. For post-processing data, xLPR-SIAM offers flexibility, scalability potential, and the possibility to develop integrated units for total risk assessments; however, an extra investment is needed to build the frame to make it accessible to programmers of different skill levels. Under a set of defined assumptions a cost estimate for NRC staff to use xLPR-GoldSim or xLPR-SIAM within the next 5 years was made using the xLPR-SIAM frame was estimated to have a higher cost over a 5-year period than using of the xLPR-GoldSim frame, due to the extra investment needed to develop SIAM to a more mature state. However, NRC staff would spend less time using the xLPR-SIAM because of the expected seamless integration with post-processing tools.

References

NRC and EPRI. "xLPR Version 1.0 Report, Technical Basis and Pilot Study Problem Results." Washington, DC: NRC, Office of Nuclear Regulatory Research; Palo Alto, California: EPRI. 2011.

GoldSim Technology Group LLC. "GoldSim Version 10.11." Issaquah, Washington: GoldSim Technology Group LLC. <<http://www.GoldSim.com>> 2011.

CONTENTS

Section	Page
ABSTRACT	ii
FIGURES	iv
TABLES	v
ACKNOWLEDGMENTS	vi
1 INTRODUCTION.....	1-1
2 CODE EFFICIENCY AND OPERATIONAL CONVENIENCE FROM END USER'S PERSPECTIVE	2-1
3 CLARITY AND READABILITY FROM AN INDEPENDENT MODEL DEVELOPER PERSPECTIVE	3-1
4 FLEXIBILITY AND ADAPTABILITY FROM A MODEL DEVELOPER PERSPECTIVE	4-1
5 FUTURE DEVELOPMENT POTENTIAL	5-1
6 CONCLUSIONS	6-1
7 REFERENCES.....	7-1

FIGURES

Figure	Page
3-1 An Example of Documentation Generated Using the Sphinx Python Documentation Generator	3-4
3-2 An Example of an Embedded and Interactive Class Diagram and Documentation Generated Using Epydoc	3-4
3-3 Example of Accessing xLPR-SIAM Documentation Generated Using Epydoc.....	3-6
6-1 Time Estimates for the Cost of Using the GoldSim and SIAM Frames in a 5-Year Period, Expressed as Probability Distribution and Cumulative Distribution Functions.....	6-8

TABLES

Table	Page
2-1 Evaluation of Input Data Access	2-2
2-2 Evaluation of Code Execution	2-3
2-3 Evaluation of Output Data Access	2-4
3-1 Code Documentation and Compatibility	3-2
4-1 Evaluation of Frame Elements for the Development of Stochastic Models.....	4-3
5-1 Comparison of Frame Features To Support Future Development.....	5-1
6-1 Estimate of the Time for GoldSim Use by NRC Staff	6-4
6-2 Estimate of the Time for SIAM Use by NRC Staff.....	6-5
6-3 Estimate of Time To Finalize the SIAM Frame	6-6

ACKNOWLEDGMENTS

This report describes work performed by the Center for Nuclear Waste Regulatory Analyses (CNWRA®) for the U.S. Nuclear Regulatory Commission (NRC) under Contract No. NRC-04-10-144. The activities reported here were performed on behalf of the NRC Office of Nuclear Regulatory Research. This report is an independent product of CNWRA and does not necessarily reflect the views or regulatory position of NRC.

The authors would like to thank T. Mintz for the programmatic review; L. Mulverhill for the editorial review; and L. Naukam for providing word processing support.

QUALITY OF DATA, ANALYSES, AND CODE DEVELOPMENT DATA

DATA: All CNWRA-generated original data contained in this report meet the quality assurance requirements described in the CNWRA Quality Assurance Manual. Each data source is cited in this report and should be consulted for determining the level of quality for those cited data.

ANALYSES AND CODES: The analyses presented in this report followed were performed using the SIAM-xLPR Version 1.0 software (Klashy, et al., 2010) and the xLPR Model Framework Version 1.0 (GSxLPRv1.02_M02) software (Mattie, et al., 2010). All analyses performed for this verification and validation report followed the CNWRA Quality Assurance Procedure-014, Documentation and Verification of Scientific and Engineering Calculations.

References

Klashy, H.B., P.T. Williams, B.R. Bass, and S. Yin. "Structural Integrity Assessments Modular-Probabilistic Fracture Mechanics (SIAM-PFM): User's Guide for xLPR." ORNL/NRC/LTR-247. Oak Ridge, Tennessee: Oak Ridge National Laboratory. 2010.

Mattie, P.D., D.A. Kalinch, and C.J. Sallaberry. "U.S. Nuclear Regulatory Commission Extremely Low Probability of Rupture Pilot Study: xLPR Framework Model User's Guide." SAND2010-7131. Albuquerque, New Mexico: Sandia National Laboratories. 2010.

1 INTRODUCTION

The U.S. Nuclear Regulatory Commission (NRC), Office of Nuclear Regulatory Research has entered into a cooperative program with the Electric Power Research Institute (EPRI) with the goal to develop a modular-based, probabilistic fracture mechanics tool or code capable of determining the probability of failure for reactor coolant system components. This code is referred to as extremely low probability of rupture (xLPR). The intention of the code is to encompass a range of physical processes, to be flexible to permit analysis of a variety of service conditions, and to be adaptable to accommodate evolving and improving knowledge. A modular design is preferred to allow for additions and modifications.

A pilot study sponsored by NRC and EPRI, and performed by Sandia National Laboratories and Oak Ridge National Laboratory was conducted as a proof of concept to test (i) the feasibility of developing probabilistic models for failure mechanisms for a component of the piping coolant system, (ii) configuration management and quality assurance for multiple-party code development, and (iii) existing tools for the prompt deployment of probabilistic models (NRC and EPRI, 2011). The problem of initiation and growth of primary water stress corrosion cracks in a dissimilar metal pressurizer surge nozzle weld of the piping coolant system in a nuclear power plant was selected as the subject of the pilot study. To test existing tools for quick deployment of probabilistic models, the commercial GoldSim[®] (GoldSim Technology Group LLC, 2011) software and open source tools [Structural Integrity Assessment Modular—Probabilistic Fracture Mechanics, or SIAM for short] (SIAM-PFM)] were identified to support a modular development of the xLPR code. GoldSim is a Monte Carlo simulation software to model dynamic systems. GoldSim has been used to model diverse systems in radioactive waste management, environmental problems, and business problems. Problems in risk analysis and reliability engineering have been addressed using GoldSim (GoldSim Technology Group LLC, 2011). SIAM has been used to deal with problems in vessel fractures, dislocation-based fracture and cleavage initiation in ferritic steels, and piping reliability including seismic events (Klasky, et al., 2010). SIAM is intended as a framework to include modern principles of probabilistic risk assessment to support the analysis of nuclear power plant safety issues (Klasky, et al., 2010). Reports on xLPR-GoldSim (Mattie, et al., 2010) and xLPR-SIAM (Klasky, et al., 2010) are available from the model developers, who discuss results of the pilot study and provide self-assessments of the merits of the GoldSim and SIAM frames.

The objective of this report is to provide an independent assessment of the GoldSim and SIAM frames to support development of the xLPR code. The evaluation focused on (i) ease of use from a user perspective, (ii) readability from a model developer perspective, (iii) flexibility and adaptability from a model developer perspective, and (iv) potential of the frame for expansion of xLPR. To develop metrics for comparison, proposals were shared and discussed with NRC staff and representatives from the nuclear power industry; feedback was elicited and incorporated into the definition of those metrics and approaches to perform the comparison. The comparison metrics were initially intended to be quantitative. However, as the evaluation proceeded, it became evident that qualitative and subjective comparisons were not entirely avoidable. For each comparison element, GoldSim and SIAM were graded according to a five-star system. For each comparison element, an expected case to which the frame was compared and graded is briefly described. The comparisons are described in Chapters 2 to 5 of this report, covering elements (i), (ii), (iii), and (iv) previously listed. The bulk of the sections is presented in the form of tables. A brief introduction precedes the tables to describe the comparison approach. Each section concludes with a summary of the main contrasts, findings, and recommendations for enhancement. Chapter 6 is a concluding analysis section informed

by findings in Chapters 2 through 5. The concluding analysis includes estimates of the time it would take for NRC staff to use the frames in the next 5 years under a set of defined assumptions. Comparing the frames using a metric of time and staff hours allows for contrast between the frames, which could be used to inform a decision of further model development.

This report includes three appendices. In Appendix A, elements of a robust quality assurance program are discussed. The configuration management system implemented to support controlled development of the xLPR codes is considered in Appendix A. Tasks to develop model validation, in the context of a quality assurance program, are provided in Appendix A. Appendix B provides results of a limited code verification of FORTRAN modules common to the xLPR-GoldSim and xLPR-SIAM codes. The verification exercise was implemented as an initial task to allow the writers of this report to become acquainted with the concepts of the xLPR pilot study. Appendix C details the approach of inserting a simple module into the xLPR-GoldSim and xLPR-SIAM codes used as the basis for the evaluation documented in Chapter 4.

2 CODE EFFICIENCY AND OPERATIONAL CONVENIENCE FROM END USER'S PERSPECTIVE

The objective of this chapter is to evaluate operational convenience of xLPR-GoldSim and xLPR-SIAM from the point of view of the end user. Three users were requested to exercise the xLPR codes, launch a multiple-realization run, and access the output data. The users had diverse backgrounds with respect to programming, ranging from very experienced to limited experience. All of the users were used to employing routinely technical software to perform analyses. The users were asked to evaluate (i) input data access, (ii) code execution, and (iii) output data access. Under (i) input data access, the users were requested to work with input parameters and evaluate the convenience of modifying values, modifying distributions, changing input parameters from constant to distributions or vice versa, and changing the distribution type from epistemic to aleatory or vice versa. With respect to the epistemic/aleatory treatment of uncertainty, the xLPR codes were designed to separate parameter value samplings associated with distributions representing epistemic (or reducible) uncertainty from samplings associated with distributions representing aleatory (or irreducible) uncertainty. This separate tracking allows expected values to be computed over the epistemic or aleatory space. In principle, only the epistemic uncertainty can be reduced with further analyses or experimental characterization; thus, this type of uncertainty is central in a decision making process. For further discussion on the epistemic/aleatory split, read the framework reports of Mattie, et al. (2010) and Klasky, et al. (2010). Under (ii) code execution, the users were requested to evaluate the convenience of launching a run, identify useful information displayed during code execution, and record the central processor unit (CPU) runtime for a run with a relatively large number of realizations (50,000). For (iii) output data access, the users were asked to evaluate the convenience of locating output for graphical display and generating such graphical displays. They were also required to evaluate the display of output statistics, the availability of tools to export data, and the convenience of those tools. The availability of multiple-realization outputs to use external tools to compute correlation matrices or other sensitivity analysis techniques was also evaluated. The xLPR codes are distributed with a couple of tools to post-process the raw output data. These codes are referred to as TRANSFORMERS and EXPECTATION. TRANSFORMERS modifies the multiple realization output data to account for detection of cracks during inspection or due to leakage detection. It is assumed that once detection occurs, the pressurizer surge nozzle weld is repaired and no more cracks develop in the repaired component. EXPECTATION is a code to compute statistics over aleatory and epistemic spaces. For example, if 1,000 epistemic realizations and 50 aleatory realizations are considered, xLPR codes execute a total of $1,000 \times 50 = 50,000$ realizations. EXPECTATION computes averages over the aleatory space and consolidates the 50,000 realizations into 1,000 epistemic realizations. Each epistemic realization is an average over 50 aleatory realizations. The 1,000 realizations represent uncertainty in the output due to the epistemic uncertainty in the inputs. For a detailed discussion of the TRANSFORMERS and EXPECTATION codes, see Mattie, et al. (2010). Both xLPR codes use these post-processing tools, but are executed in a different manner. The convenience of execution of these codes and the availability of the output data were also considered in the evaluation in this chapter.

The results for (i), (ii), and (iii) are summarized in Tables 2-1 to 2-3. The third column in these tables represents an *expectation* used as reference to assign a star grade. A five-star system was selected, and stars were deducted depending on whether a code was deemed to fall short of the *expectation*. The tables identify and briefly discuss strengths and shortcomings.







Table 2-1. Evaluation of Input Data Access		
xlPR-GoldSim	xlPR-SIAM	Expectation
Available Options for Changing Input Values		
 <p>Strengths: Simple Microsoft® Excel® and GoldSim® dashboard interface. The Excel interface is convenient to perform extended functions: (e.g., Excel files can be searched and filtered by keywords, thus facilitating the location of input parameters, or sorting input parameters by attributes. Runs can be launched at specified quantiles of the input parameters using GoldSim PRO.</p> <p>Limitations: A recent version of Excel is required for the model to run. Excel is not accessible while GoldSim is running, except by using workarounds. Changing distribution type (e.g., from uniform to normal) requires model changes. The same simulation setting needs to be defined in more than one place in a few instances. The input files for TRANSFORMERS and EXPECTATION are manually constructed, which is error prone if a nondefault case is executed.</p>	 <p>Strengths: Simple interface with data classified in tabs and options provided by pull-down menus in a GUI. The GUI fully controls relevant inputs for post-processors (TRANSFORMERS and EXPECTATION), which avoids user error.</p> <p>Limitations: There is a limited set of distribution functions for sampling. The use of location-scale-shape parameters is nonintuitive (e.g., a uniform distribution is specified by the location and scale parameters). There is no option for generating standard deterministic inputs (e.g., all sampled parameters fixed at given quantile or mean values). Such an option is convenient for testing. Default input values are hard-wired. The time steps are hardwired. Some bugs are present in the data input.</p>	<p>Intuitive data entry interfaces and simple input change provisions</p> <p>The user can change the majority of parameters and default inputs</p> <p>Common distribution functions are available</p> <p>Binary correlations can be specified for correlated sampling</p> <p>Simple to generate standard deterministic inputs (e.g., all sampled parameters fixed at given quantile or mean values)</p> <p>Simulation settings are specified in unique fields</p> <p>Specification files for post-processing are automatically generated</p>
Changing From Constant to Distribution or Vice Versa		
 <p>Limitations: Modifications to the model file are needed to change a parameter from constant to stochastic or vice versa. Deleting rows in the input data Excel spreadsheet requires redefining the Excel cell references in the corresponding GoldSim element (changes are cumbersome).</p>	 <p>Simple to change from stochastic (sampled parameter to constant parameters) or vice versa</p>	<p>Simple to change from stochastic (sampled parameter to constant parameters) or vice versa</p>
Changing an Epistemic Parameter to Aleatory or Vice Versa		
 <p>Simple switch offered in the Excel file to change from aleatory to epistemic, or vice versa GUI = graphical user interface</p>	 <p>Simple pull-down menu provided to switch from aleatory to epistemic or vice versa</p>	<p>Simple to change any parameter from aleatory to epistemic or vice versa</p>





Table 2-2. Evaluation of Code Execution		
xlPR-GoldSim	xlPR-SIAM	Expectation
Launch a Run  <p>Simple to set input flags (through the dashboard). Icon available to launch a run. Run can be paused if needed and restarted easily. There is an option for launching a run from the command line.</p>	 <p>Well-organized run-execution tab. Launching a run is straightforward. There is an option for launching a run from the command line.</p>	<p>Launching a run is intuitive and simple</p>
Information Display During a Run  <p>Strengths: Polished display of run information. The elapsed time for a run is saved with the model file. Dashboard elements used to display some indicators (e.g., Cracks Formed, Cracks Coalesced, Surface Cracks, Through Wall Cracks, Leak Rate). There is a green button in the dashboard that signals a pipe rupture event and the time of the event. Result elements can display results for a realization in real time, if they are opened before the run is launched.</p> <p>Limitations: The information displayed by realization is constrained if the model is launched in parallel processing mode</p> <p>CPU Time for 50,000 Realizations (Hours) 10:17 hr + 9:20 hr for data export 6:17 hr for parallel run on two CPUs 5:15 hr + 2:04 hr for data export</p> <p>5:08 hr + 2:04 hr for data export 1:42 hr for parallel run on four CPUs 6:41 hr + 2:00 hr for data export</p>	 <p>Strengths: Output legible. Screen outputs designed to diagnose problems.</p> <p>Limitations: No automatic saving of the runtime in a readily accessible location/interface that is persistent (saved with the project file results). No display of time of launch and elapsed time that is visible during the entire run.</p>	<p>Display information to monitor a run: execution time, realization number, error messages and warnings</p> <p>Key information is saved with the model file or in log files</p>
Computer Specs		
5:38 hours (run + data export)		Windows XP, Intel Core Duo CPU @ 2.67 GHz; RAM 2GB; 32-bit
3:39 hours (run + data export)		Windows 7, Intel® Core™ i7 CPU 860 @ 2.8 GHz; RAM 4GB; 32-bit
3:46 hours (run + data export)		Windows 7, Intel® Core™ i7 CPU 870 @ 2.93 GHz; RAM 4GB; 32-bit
4:36 hours (run + data export)		Windows XP, Intel Core2 duo, 3 GHz; RAM 3GB; 32-bit







Table 2-3. Evaluation of Output Data Access		
xlPR-GoldSim	xlPR-SIAM	Expectation
Locating Data for Graphic Display		
 <p><u>Strengths:</u> Relevant output data are tracked in a dashboard and logically organized. Elements in the model provide additional data for graphic display, which can be retrieved after execution, if needed. GoldSim elements can be "checked" to store multirealization data.</p>	 <p><u>Strengths:</u> The results are conveniently grouped after execution of the post-processing tool.</p>	Graphic elements can be easily located Results are logically grouped
Generating a Graphic Display		
 <p><u>Strengths:</u> Generating plots is straightforward, with the click of a button or a results element. Graphic displays provide complete control of the plot appearance. Convenient options are provided to create graphical files (e.g., jpg).</p>	 <p><u>Strengths:</u> Generating plots is straightforward, via a pull-down menu and clicking a refresh button. An option is provided to create a graphical file in PDF format.</p> <p><u>Limitations:</u> There are no controls for changing plotting options, other than a switch from linear to logarithmic scale. The user must select the refresh option to update plot when a new parameter is selected from a pull-down menu. SIAM graphic display depends on outputs from TRANSFORMERS and EXPECTATION, which may not be defined in case of a deterministic output. Plots from deterministic runs are not available.</p>	<p>Graphic displays automatically generated for selected outputs with minor user action</p> <p>Graphic displays are available for probabilistic and deterministic outputs. For probabilistic displays, statistics (e.g., quantiles or means) can be displayed</p> <p>Controls are provided to change appearance, add labels, and switch between linear and logarithmic scales</p> <p>Legends for plots with multiple curves are clear</p> <p>Options are provided to export plots in a graphic format file (e.g., jpg, tiff, pdf)</p>
Displaying Statistics (Mean, Quartiles)		
 <p><u>Strengths:</u> Efficient display of statistics for raw (combined aleatory and epistemic) data. Adequate control on the appearance of the statistic curves and confidence intervals.</p> <p><u>Limitations:</u> Computation of separated statistics over aleatory and epistemic spaces requires post-processing tools. Third-party software is needed to read output files from post-processing and graphic display.</p>	 <p><u>Strengths:</u> Statistical displays are available. The graphic displays use outputs from TRANSFORMERS and EXPECTATION post-processing. Thus, the graphic displays show statistics over the epistemic space.</p> <p><u>Limitations:</u> Horse-tail plots are not available. Graphic displays over the raw parameter space (combined epistemic and aleatory) are not available. It takes a long time for graphic displays to be generated (however, files containing precomputed statistics are small in size). It appears that statistics are computed on demand from the EXPECTATION epistemic output files. There are no plots for distributions or cumulative distributions of single-value outputs (a template was provided in Excel to plot results, which is not very useful given the significant user interaction required and that it does not account for the epistemic-aleatory sampling scheme).</p>	<p>For probabilistic runs, plots show mean, quartiles, and individual realizations (horse-tail plots)</p> <p>Plots are generated in a reasonable time</p> <p>Plots show meaningful statistics (i.e., statistics split over aleatory or epistemic space, or raw data statistics)</p> <p>Plots (probability distribution or cumulative distributions) are available for single-value outputs from multiple realizations</p>

Table 2-3. Evaluation of Output Data Access (Continued)		
xLPR-GoldSim	xLPR-SIAM	Expectation
Tools To Export Data		
★★★★★	★★★★★	Data are conveniently exported from graphic displays
Exported Data Format		
A button is available to export data as a text file	A button is available to export data as a text file	
★★★★★	★★★★★	
Data are exported in matrix form that can be directly imported to Excel or other analytical software. No proprietary formats are used.	<p>★★★★★</p> <p>Data exported in text form that can be imported into Excel. No proprietary formats are used.</p> <p><u>Limitation:</u> The format of the output data needs additional manipulation to be plotted in external software. Data from plots containing multiple series are listed in the form of blocks (x,y) as opposed to a matrix with multiple columns.</p>	Format of output data is convenient to be imported to other plotting or analytical software (e.g., Excel, Matlab, Mathematica)
Availability of Data To Support Sensitivity Analyses		
★★★★★	★★★★★	<p>Data needed for computing correlation matrices, performing regression analyses, or conducting other external sensitivity analysis are readily available (e.g., the values of sampled input parameters and multiple realization outputs)</p> <p>Data are available to support sensitivity analyses over the epistemic space</p>
<p>GoldSim includes tools to conduct sensitivity analyses and compute correlation coefficients. Correlation matrices would have limited value given the aleatory-epistemic split.</p> <p><u>Limitations:</u> Data are available to support external sensitivity analyses; however, the model file must be adjusted to get the input sampled values and to parse the input parameters into epistemic or aleatory sets. Such adjustments to the model file can be cumbersome.</p>	<p>Strengths: Sampled parameter values are separated into aleatory and epistemic sets. This information, combined with information from the EXPECTATION tool allows performing sensitivity analyses over the epistemic space.</p>	

Discussion and Recommendations

From the end-user perspective, there is no high contrast in the ranking of the codes with respect to the considered star systems. xLPR-GoldSim and xLPR-SIAM codes present different advantages and disadvantages. xLPR-GoldSim provides polished interfaces and plotting options. However, xLPR-GoldSim lacks flexibility to perform a number of actions (e.g., embedded post-processing, limitations to manipulate inputs, cumbersome to extract data for sensitivity analyses). The runtime for xLPR-GoldSim is at least twice as long compared to xLPR-SIAM, accounting for time needed for data export. The runtime can be shortened significantly by taking advantage of parallel processing spanning the realizations among up to four CPUs using GoldSim Pro (more than four processors can be enabled by GoldSim's Distributed Processing Module, at an additional cost). The total time (runtime plus data export time), accounting for distributed processing in several CPUs, is at best comparable to the xLPR-SIAM runtime. Note that a number of factors seem to affect the total runtime in xLPR-GoldSim, including the amount of RAM memory, CPU frequency, hard drive spin frequency, and the hard drive hardware interface (IDE or SATA). The data export time can take several hours, and parallel processing cannot be used to reduce it. Other strategies could be designed to capture outputs during runtime using dynamically linked libraries to address the lengthy data export time.

xLPR-SIAM offers flexibility, but its graphical user interface (GUI) is less polished and it lacks plotting options. In principle, all of the xLPR-SIAM limitations noted in Tables 2-1 through 2-3 can be addressed with programming effort. In contrast, the xLPR-GoldSim limitations are mostly intrinsic to the GoldSim frame and may not be solved except by changes by GoldSim developers.

General Recommendations

During this evaluation, it was noted that both codes implement only binary correlations for input parameters. None of the codes have the capability to sample ternary or higher correlation systems. For example, to completely define a system with three correlated variables, A, B, and C, the following correlations need to be specified: correlation (A, B), correlation (B, C), and correlation (A, C). However, one and only one of these correlations can be currently specified in the xLPR codes; thus, the definition of correlations is incomplete and it is not possible to properly generate correlated samples. Consideration could be given to other sampling algorithms that provide flexibility to fully specify correlation or covariance matrices. Sandia's Latin Hypercube Sampling Software (Swiler and Wyss, 2004) is an available tool that allows for complete definition of covariance matrices. This code is distributed as part of the DAKOTA Project (dakota.sandia.gov), which also includes other tools for optimization, parameter estimation, and sensitivity analyses. The current release of DAKOTA is Version 5.1 (Adams, et al., 2010).

The specification files for the TRANSFORMERS and EXPECTATION tools (i.e., options.txt and EXP_options.txt, respectively) require a different number of text rows depending on the case considered. This type of approach is inconvenient because the user is forced to consult the user guide to make sure that the appropriate entries were provided in the input files options.txt and EXP_options.txt. Instead, it is recommended to fix the number of input lines in these files, properly labeling the entries. It should be clear to the user that some of the inputs are used only if needed by the case under consideration. Currently, xLPR-SIAM generates the appropriate files options.txt and EXP_options.txt from data input in the GUI. xLPR-GoldSim, on the other hand, needs manual modification of those files, which is error prone. The errors can be

minimized by fixing the number of lines in the input files options.txt and EXP_options.txt. Also, some entries in these files could be controlled by entries in the dashboards (e.g., number of stochastic and aleatory realizations) and dynamically linked libraries (DLL).

The output files from EXPECTATION do not include headers. Readability of these output files will be enhanced by the addition of headers. In particular, in the statistics files (files with a _stat.txt suffix), the inclusion of headers to properly identify columns associated with mean values and quantiles will help the user eliminate the extra step of consulting the user guide and additional input file defining the quantiles (i.e., quantiles.txt). Also, a standard deviation column is recommended to be added to the statistics files.

xLPR-GoldSim Recommendations

- Allow some control of input files to the TRANSFORMERS and EXPECTATION codes from the GoldSim dashboard, via DLLs. This approach would minimize user error.
- Consider developing a DLL to capture realization data to alleviate the problem of extended export time. For example, a time-series output can be captured into a text file, after a realization is completed. A time series per realization would be appended to the text file as a new row of data. This approach can be an efficient alternative to data exporting. Such an approach may not significantly decrease the code efficiency and yet will bypass the need for manual data export. The disadvantage of this approach is that it does not work for parallel runs spanning several processors. Also, the matrix of data in the text files would need transposing before use in the TRANSFORMERS code.

xLPR-SIAM Recommendations

- Expand the set of distributions for sampling. Currently, only a small set of distributions is available.
- Provide an option to generate standard deterministic inputs (e.g., all sampled parameters fixed at given quantile or mean values).
- Explain in the user guide how scale-shape-location parameters map to standard parameters to define distributions.
- Provide information on the total run time while the run is executed. Save the total execution time in a log file.
- Allow for plots (cumulative distribution and probability density) of single-value outputs per realization to eliminate the need of the Microsoft® Excel® template file. Perform appropriate statistics for the single-value outputs accounting for the epistemic-aleatory sampling scheme.
- Expose plotting options. The graphics appear to be generated using the Qwt (Rathmann, 2011) library [with PyQwt (Vermeulen, 2011)], which is a library with a range of plotting options.
- Allow for automatic refresh of plots (e.g., when different data are selected from the pulldown menu).

- Allow for plotting of deterministic outputs.
- Include horse-tail plots and plots of raw data.
- Enhance the format of output data (matrix with multiple columns) exported from the visualization results window.

3 CLARITY AND READABILITY FROM AN INDEPENDENT MODEL DEVELOPER PERSPECTIVE

This section is aimed at enabling testers who are experienced programmers to comment on clarity and readability of the source codes to support code maintenance. Programmers with variable knowledge of GoldSim, Python, and FORTRAN were requested to study xLPR source codes and record the time needed to gain enough familiarity with the xLPR algorithms to implement basic changes to the model. Three programmers were asked to study the source codes of the xLPR-GoldSim to attempt specific model modifications, detailed in Chapter 4. Out of the three programmers, only one was able to implement those modifications within a reasonable amount of time (a maximum time was set *a priori* to avoid project overruns in schedule and budget). This programmer succeeded within the limited time because he had previous knowledge of both GoldSim and Python (main language for SIAM). The successful programmer estimated that it took him approximately 25 hours to gain basic familiarity with xLPR-GoldSim to initiate implementation of model changes. For xLPR-SIAM, the programmer estimated 57 hours would be needed to gain enough familiarity to start working on the source code. The programmer attributed the longer familiarity time for xLPR-SIAM to the need to work with two languages (FORTRAN and Python) and several Python libraries [e.g., PyQt (Riverbank Computing Limited, 2011)]. On the other hand, xLPR-GoldSim only needs some familiarity with GoldSim and FORTRAN languages. The other programmers indicated it would take them much longer because they needed to gain required knowledge of FORTRAN, Python, Python libraries, and GoldSim before comfortably implementing changes to the codes. Estimates of the time needed to gain familiarity with the codes are recorded in Chapter 4 and also used in the cost computations in Chapter 6.





The programmers were also asked to evaluate (i) existing internal documentation, (ii) capability of the frame to incorporate and retrieve internal documentation, (iii) operating system compatibility, and (iv) programming language compatibility (GoldSim or SIAM). Evaluation results are recorded in Table 3-1. A five-star system similar to Chapter 2 is adopted in this chapter. The *Expectation* column defines the attributes to assign the star grade.

Discussion and Recommendations

xLPR–GoldSim

The xLPR-GoldSim model is readable (i.e., it is possible to follow the model logic and the computational sequence). xLPR-GoldSim is visually organized in a manner that the flow of information becomes clear. The readability is facilitated by visual aids (e.g., customized icons, graphics, and influence arrows), the “function of” and “affects” view trees to identify GoldSim element dependencies, as well as the various browsing capabilities (e.g., GoldSim allows browsing of elements by type). Many options are available to provide internal documentation: text boxes, text per GoldSim element, hyperlinks, visual elements (diagrams/drawings/plot), and customized icons. Extra effort should be aimed at taking advantage of internal documentation capabilities, describing the action of GoldSim elements, and briefly explaining the computations at the GoldSim container level. A GoldSim container is a box icon that groups a number of computations in a single workspace.

Note that model readability was compromised in some instances by adapting the limited GoldSim elements to perform specialized functions. For example, dealing with arrays to separately track cracks and their properties, such as crack length, depth, and type (i.e., surface

Table 3-1. Code Documentation and Compatibility		
xlPR-GoldSim	xlPR-SIAM	Expectation
Existing Internal Documentation		
 <p>Strengths: "Function of" and "affects" native GoldSim functions, influence arrows, and the visual organization of model elements facilitate understanding of the flow of information, with minimal extra documentation needed. Frame allows browsing GoldSim elements by type.</p> <p>Limitations: Minimal internal documentation (description fields by element are minimally populated) is available. In general, documentation capability was not taken advantage of.</p>	 <p>Limitations: Only few class/function docstrings have full entries. Examples of the use of functions are not provided. Use of "Help" option for browsing "SIAM-xlPR API" documentation for viewing class and module hierarchies is available, but the available information is limited. The usefulness of SIAM-xlPR API documentation can be greatly improved by populating docstrings and taking advantage of options available from the API documentation-generating software.</p>	<p>This code is well documented internally, with comments allowing the programmer to understand the action of functions and the flow of computations and information.</p> <p>The code includes enough comments either embedded in the model or located in the electronic file structure to understand the flow of information within the source code.</p>
Capability of Frame To Incorporate and Retrieve Internal Documentation		
 <p>Strengths: Multiple options are available to provide internal documentation: text boxes, text per GoldSim element, hyperlinks, and visual elements (diagrams/drawings/plots, customized icons). The availability of influence dependencies ("affects" and "function of") greatly facilitates understanding the computations, even if minimal developer documentation was available.</p>	 <p>Strengths: Multiple options are available to provide internal documentation: HTML generation, docstrings, and source code comments. Open source, third-party programs and modules are available to enhance development of documentation accessibility (e.g., the Epydoc or SPHINX python documentation generator)</p> <p>Limitation: Extra learning is needed to use documentation tools.</p>	<p>Multiple options are available for internal documentation.</p> <p>Internal documentation is in a format that is easy to read and understand. Internal documentation is searchable and can be queried.</p> <p>Maintenance of internal documentation is simple, and tools to maintain internal documentation are simple to learn.</p>
Operating System Compatibility		
GoldSim is compatible with Microsoft® Windows® operating systems. Different behavior in regard to response to hyperlinks embedded in dashboard buttons was noted in Windows XP and 7.	<p>xlPR-SIAM is compatible with Windows operating systems. Difficulties were encountered during installation for some machines; however, we were unable to diagnose the cause.</p> <p>There was partial success in installing xlPR-SIAM in Linux. It was not possible to launch the GUI to execute the code. However, the code was successfully executed from a command line, after compilation in Linux. Making SIAM fully compatible with Linux will require debugging and development effort.</p>	Compatible with several operating systems
Programming Language Compatibility		
Interface tools are available to incorporate functions programmed in FORTRAN, C, and C++. Minor extra interface code is needed for proper dialog with GoldSim. GoldSim can also link to Excel® and use computations driven by Excel, including extended Visual Basic® functions.	Interface tools are available to incorporate functions programmed in FORTRAN, C, and C++. Minor extra interface code is needed for proper dialog with Python.	Interface available to establish dialog with functions and subroutines programmed in FORTRAN, C, and C++. Minimal code changes are needed to make a code communicate with the frame. Compilation of code is simple.

crack or through wall crack), required a clever use of looping containers, elements to trigger discrete changes, and integrator elements. In a traditional programming language, such as FORTRAN, dealing with arrays and array updates is more transparent. For example, an array index can represent a crack and the appropriate array entry can be directly accessed within a loop block. GoldSim can be adapted to perform different functions and workarounds can be implemented using existing GoldSim elements. As models grow in complexity, however, these workarounds tend to compromise model readability and transparency. The final outcome, in our opinion, is that a complex GoldSim model requires a dedicated custodian with complete familiarity of the model and understanding of the workarounds to provide maintenance to the model.

xLPR-SIAM

The xLPR-SIAM is difficult to follow, in general. Some model components (e.g., timeloop module) are well documented, have a linear sequence, and are easy to understand. For example, the main sequence of computations within a realization is controlled by a FORTRAN subroutine referred to as the timeloop module, which is of a straightforward structure. In fact, some of the programmers used the FORTRAN timeloop as a reference to decipher the computational sequence in xLPR-GoldSim. For some other components, the information flow sequence is not as transparent (e.g., Python code for data management and definition of GUIs). Part of the difficulty in properly reading the source code is that familiarity with Python, FORTRAN, and Python libraries to develop GUIs is required. An object-oriented structure was selected to write the Python code, while FORTRAN follows a traditional functional and subroutine approach. The internal documentation has variable levels of detail. The FORTRAN timeloop module was found to be internally documented with an appropriate level of detail. The Python objects, on the other hand, could include additional documentation and examples to facilitate an independent programmer's understanding of the intended flow of information.

Using the Python programming language could significantly enhance the internal documentation of Python functions. The standard language includes a feature for defining documentation entries for classes, methods, functions, and modules in documentation strings (commonly referred to as docstrings). Docstrings can incorporate descriptions to explain the purpose and usage of the code, as well as provide examples. The docstring entries can be accessed by default from the Python interactive interpreter where formatting of the docstring is automatically handled by the pydoc module (Python Software Foundation, 2011a). The pydoc module is provided in the Python standard library and can also be used to generate an HTML version of the documentation for viewing in a web browser. In addition to pydoc, the doctest module (also included in the standard library) (Python Software Foundation, 2011b) can utilize properly formatted code usage examples included in docstrings to perform automated testing of the source code.

Use of open source, third-party tools like Epydoc (Loper, 2011) or Sphinx (Brandl, 2011) can greatly enhance information from docstrings and improve documentation accessibility. Both documentation generation tools can take advantage of markup text to generate enhanced formatting and hyperlinks, as well as provide convenient documentation search capabilities for the end user. The Epydoc tool can also be used to auto generate class diagrams that are embedded in the HTML documentation and contain hyperlinks that permit the user to navigate documentation by clicking directly on the class diagrams. Figure 3-1 is a screen capture of documentation generated using Sphinx that can be read using an internet browser. The screen capture shows highlighted Python code, and a quick search field, which are automatically generated by the Sphinx functions. Figure 3-2 is a screen capture showing an

Python v2.6.6 documentation » The Python Standard Library »

previous | next | modules | index

Table Of Contents

2. Built-in Functions

3. Non-essential Built-in Functions

Previous topic

1. Introduction

Next topic

4. Built-in Constants

This Page

Report a Bug

Show Source

Quick search

Enter search terms or a module, class or function name.

2. Built-in Functions

The Python interpreter has a number of functions built into it that are always available. They are listed here in alphabetical order.

abs(*x*)

Return the absolute value of a number. The argument may be a plain or long integer or a floating point number. If the argument is a complex number, its magnitude is returned.

all(*iterable*)

Return True if all elements of the *iterable* are true (or if the iterable is empty). Equivalent to:

```
def all(iterable):
    for element in iterable:
        if not element:
            return False
    return True
```

New in version 2.5.

any(*iterable*)

Return True if any element of the *iterable* is true. If the iterable is empty, return False. Equivalent to:

```
def any(iterable):
    for element in iterable:
        if element:
            return True
    return False
```

New in version 2.5.

Figure 3-1. An Example of Documentation Generated Using the Sphinx Python Documentation Generator (Note Text Highlighting for Python Code and Search Feature). Screenshot Captured From Python Documentation (Python Uses Sphinx) Website (Python Software Foundation, 2011c).

Home Trees Indices Help epydoc 3.0.1

Package epydoc :: Module apidoc :: Class VariableDoc

[hide private](#)
[frames](#) | [no frames](#)

type **VariableDoc**

[source code](#)

```

classDiagram
    class APIDoc {
        docstring: string or None
        docstring_lineno: int
        other_docs: bool
        metadata: (str, str, ParsedDocstring)
        docs_extracted_by: str
        __init__(self, **kwargs)
        __setattr__(self, attr, val)
        __repr__(self)
        apidoc_links(self, **filters)
    }
    class VariableDoc {
        name: str
        is_imported: bool
        is_instvar: bool
        is_alias: bool
        is_public: bool
        defining_module
        __init__(self, **kwargs)
        __repr__(self)
        apidoc_links(self, **filters)
        is_detailed(self)
    }
    class ValueDoc
    class DottedName
    class DocstringField
    class ParsedDocstring

    APIDoc <|-- VariableDoc
    VariableDoc o--> ValueDoc : container
    VariableDoc o--> DottedName : value
    VariableDoc o--> ParsedDocstring : type_descr
    VariableDoc o--> DocstringField : descr
    VariableDoc o--> DocstringField : summary
    VariableDoc o--> DocstringField : extra_docstring_fields
    
```

API documentation information about a single Python variable.

Note: The only time a VariableDoc will have its own docstring is if that variable was created using an assignment statement, and that assignment statement had a docstring-comment or was followed by a pseudo-docstring.

Instance Methods

__init__(self, **kwargs)

Construct a new APIDoc object.

[hide private](#)
[source code](#)
[call graph](#)

Figure 3-2. An Example of an Embedded and Interactive Class Diagram and Documentation Generated Using Epydoc (Loper, 2011)

embedded interactive class diagram. However, enhancing the documentation requires learning additional tools and text markup syntax and can increase documentation maintenance.

Note that xLPR-SIAM has used the Epydoc tool for generating its code application programming interface documentation. xLPR-SIAM provides a Help menu item in its GUI interface that accesses the system web browser to display HTML documentation generated with Epydoc and included in the xLPR-SIAM distribution. An example of the Help menu item and the HTML documentation is provided in Figure 3-3. The documentation provided includes a set of framed windows with hyperlinks for viewing class and module hierarchies, docstring content, and module source code. However, the docstrings are minimally populated, and therefore, the information displayed in the Help browser is of limited value. Further, the authors have not taken advantage of auto generation of class diagrams or use of markup text. xLPR-SIAM does not make extensive use of the testdoc module.

A comment is made with respect to operating system compatibility of xLPR-SIAM. Python and the xLPR-SIAM dependencies were installed on a Linux machine, Debian 5 distribution. The process to install the dependencies required several hours, due to the need for (i) finding and downloading libraries, (ii) compiling and installing the dependencies, and (iii) compiling FORTRAN codes to be invoked by Python using the f2py utility.¹ Additional dependencies are needed (e.g., Berkeley database and Python bindings) in the Linux installation. Python needs to be uninstalled and compiled again after installation of the Berkeley database; otherwise, Python database handling functions are not enabled. After a process that took several hours, the SIAM GUI could not be made to work. However, xLPR was launched from a Python command line, using as input a database file generated in Microsoft® Windows®. The run was successfully completed; however, it yielded different results than the Windows run. Some data inputs were not properly read from the input database. For example, the Linux run recorded fewer stochastic variables than the Windows run. It appears that xLPR-SIAM can be made to run in Linux, but additional effort and debugging is needed. On the other hand, GoldSim is only designed to work with Windows operating systems.

Recommendations

For both xLPR codes, the recommendation is to enhance the internal documentation. For xLPR-GoldSim, it is recommended to enhance documentation by explaining the purpose of key GoldSim model elements, and to use visual aids and text boxes to explain actions at the container level (minimal, to the point documentation will be sufficient to facilitate the learning effort).

For xLPR-SIAM, it is recommended to enhance the documentation by consistently populating docstrings, provide examples for the use of functions in these docstrings, and more thoroughly utilize capabilities of available Python documentation tools for Python functions. An interactive diagram of the class structure would also facilitate the learning effort.

¹ f2py is a FORTRAN to Python interface generator that comes with Python. FORTRAN code can be compiled using f2py so to make FORTRAN code functions or subroutines executable from a Python command line or within Python code.

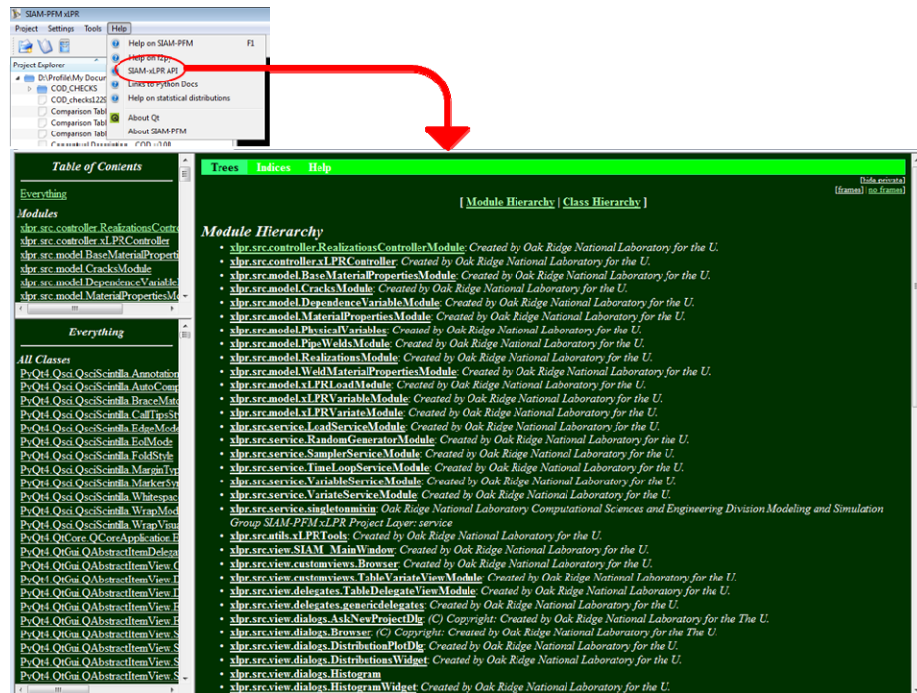


Figure 3-3. Example of Accessing xLPR-SIAM Documentation Generated Using Epydoc

4 FLEXIBILITY AND ADAPTABILITY FROM A MODEL DEVELOPER PERSPECTIVE

The objective of this chapter is to evaluate the flexibility of the frames (GoldSim and SIAM) to allow for code maintenance and code modifications. To explore the model source code and structure in more detail, “dummy” modules were inserted in both versions of xLPR to perform the same actions. The following outline defines the approach adopted to implement the dummy module:

- Develop a FORTRAN procedure
 - Compile the procedure into a DLL for use with GoldSim, or compile with the f2py utility¹ for use in SIAM
 - The procedure takes as input two random factors, r_1 and r_2 , used to modify the crack depth and crack length
 - The depth and length are provided as inputs to the procedure
- Define the parameters r_1 and r_2 as random
 - Input these random parameters in the master spreadsheet in the case of GoldSim or in one of the GUI input parameter tabs in SIAM
 - Allow these random parameters to follow available distributions in xLPR (e.g., normal, uniform, log distributions)
 - Allow r_1 and r_2 to switch from aleatory uncertainty to epistemic uncertainty, and from one distribution to another
- Apply the random factors, r_1 and r_2 , after the subroutine to compute crack growth
 - Apply the factors to all active cracks
 - Ensure that physical bounds (e.g., thickness or diameter) are not exceeded
- Track r_1 and r_2 in the set of appropriate random input parameters (epistemic or aleatory)
- Make sure these new input parameters are added to appropriate elements in GoldSim, output files, or databases

Based on the incorporation of the dummy module, programmers were asked to evaluate the convenience of tools available in the frame for the development of stochastic models. From the CNWRA experience in development of performance assessment models to deal with environmental problems and problems in radioactive waste management, it was deemed that a flexible frame should include defined tools or functions to construct Monte Carlo models. For example, a frame should include functions to (i) sample input parameters from distributions,

¹ f2py is a FORTRAN to Python interface generator that comes with Python. FORTRAN code can be compiled using f2py so to make FORTRAN code functions or subroutines executable from a Python command line or within Python code.

(ii) record sampled values from multiple realizations, (iii) manage outputs from multiple realizations, and (iv) create graphic display. The testing performed in this section was aimed at evaluating the effectiveness of the frame functionality for the efficient deployment of stochastic models. In developing stochastic models, a number of intermediate outputs are commonly tracked to understand the flow of computations and their influence on the main outputs. Therefore, an additional task was performed to evaluate (v) the effort needed to expose and record an additional parameter output.

Programmers engaged in this task were asked to estimate the time needed to study GoldSim, Python, and Python libraries, and to have enough knowledge to develop stochastic models or perform model maintenance. The responses to this question varied. A programmer with previous familiarity in GoldSim recorded that it took him 17 hours to get properly acquainted to proceed with the dummy module exercise. Other programmers without any previous GoldSim experience estimated it would take them up to 120 hours of GoldSim study to gain enough knowledge for model development and model maintenance. For SIAM, a programmer with previous familiarity with Python recorded that it took him 32 hours of preliminary study to gain basic knowledge of SIAM before undertaking the dummy module exercise. Other programmers without previous exposure to Python estimated a range from 80 to 180 hours to become acquainted with Python and Python libraries to be able to understand SIAM's source code. Note that although the other programmers for both cases were not experienced in GoldSim or Python, they were experienced engineers with programming backgrounds in FORTRAN.

To control the project scope and budget, programmers were given approximately 2 weeks to work on the dummy module per frame. In general this time was not adequate to gain enough knowledge on the frame and on the xLPR architecture to enable programmers to implement the module. Only the programmer with previous background in GoldSim and Python was able to successfully implement the dummy modules within those time constraints.

The successful programmer recorded that it took 78 hours total to implement the dummy module into xLPR-GoldSim. This total time included the effort to study GoldSim and FORTRAN and become acquainted with the xLPR-GoldSim architecture. The initial review of background material was spent mostly in learning FORTRAN basics and DLL interfacing. Additional time was spent learning the flow of information and the computational sequence of the model. The 78 hours were approximately evenly spent reviewing background material, studying the model to identify changes, and implementing changes.

The programmer recorded a total of 92 hours to implement the dummy module into xLPR-SIAM. This time includes effort required to review programming languages and module dependencies (e.g., Python, FORTRAN, PyQT), review the xLPR-SIAM framework source code, and integrate the dummy module into the framework. The programmer did not have background experience in PyQT. The programmer spent most of the time studying PyQT, the model Python class structures, and PyQT GUIs for data input and output. The computational sequence is fully contained in the FORTRAN timeloop procedure, which is a relatively short code with intuitive logic. Limited time was spent learning to compile a FORTRAN module into a Python object and learning the FORTRAN variable encapsulation. When all necessary additions to the source code were identified, actual implementation of changes in the code was straightforward and took significantly less time than the time required to study the source code.

Table 4-1 presents the results of the evaluation of the frame elements to support development of stochastic models.





Table 4-1. Evaluation of Frame Elements for the Development of Stochastic Models		
xLPR-GoldSim	xLPR-SIAM	
Sample Input Parameters	Expectation	
 <p><u>Strengths:</u> GoldSim® offers random and LHS sampling strategies. A large variety of distribution types are available within a stochastic element.</p> <p><u>Limitations:</u> The xLPR-GoldSim architecture requires duplicates of stochastic parameters to account for the epistemic-aleatory split. To add a new stochastic parameter, changes in several locations and logic to select the appropriate value are needed.</p>	 <p><u>Strengths:</u> A variate class was developed so that a number of tasks could be directly accomplished with few lines of code. Once a variable is declared as an object of the variate class, xLPR-SIAM can automatically display the variable name in a tab in the GUI (with all of the pulldown menus available), the variate values are tracked in appropriate log files, and plots for sampled values are available.</p> <p><u>Limitations:</u> Parameter names are hardcoded. The developer must determine suitable storage location for the parameter instance in the PipeWeid class as well as understand the correct syntax for defining the variate object.</p>	<p>Variables can be easily defined as stochastic.</p> <p>Sampled values are easily accessible to be used within the sequence of computations.</p> <p>Switching between the available distribution functions is simple.</p> <p>The process to create a sampled variable can be made systematic.</p>
Record Input Parameters	Expectation	
 <p><u>Strengths:</u> Input specifications for parameters are retained in the linked Microsoft® Excel® spreadsheet. Saving of sampled values for input variates can be easily toggled "on" or "off" in GoldSim.</p> <p><u>Limitations:</u> xLPR-GoldSim allows exporting of sampled input values into external files. However, the model architecture requires workarounds to consolidate sampled values into epistemic and aleatory files. Separation of the input parameters into epistemic and aleatory is needed to properly compute correlation statistics and use other sensitivity analysis techniques.</p>	 <p><u>Strengths:</u> Input values are automatically recorded to a project database file as part of class objects. Input constants and sampled parameters are recorded in external text files. The format of the output files tracking the epistemic and aleatory parameters is a simple matrix format (columns associated with different parameters, rows associated with different realizations) with clear labels.</p>	<p>Elements and frame functions are available to keep track of sampled values for a particular realization.</p> <p>A few lines of code or minimal effort is sufficient to keep track of sampled values.</p> <p>Sampled values can be conveniently retrieved for sensitivity analyses.</p> <p>The process to keep track of sampled values can be made systematic and standard.</p>







Table 4-1. Evaluation of Frame Elements for the Development of Stochastic Models (Continued)		
xLPR-GoldSim	xLPR-SIAM	Expectation
<p>Manage Output from Multiple Realizations</p> <p></p> <p><u>Strengths:</u> GoldSim automatically performs data management. Few steps are required to keep track of multiple-realization output.</p>	<p></p> <p><u>Limitations:</u> Timesteps are hard coded. The output from multiple realizations is not managed internally in the code. To keep track of output time series, it is necessary to create independent text files. Creation of text files to capture additional intermediate outputs (time series or single-value outputs) per realization requires multiple steps (e.g., declaring arrays, recompiling the FORTRAN time loop, declaring the newly added text file as input for the post-processors TRANSFORMERS and EXPECTATION). Some steps need customized Python code and customized changes to the FORTRAN timeloop. Depending on which output is exposed, distinct changes are needed in the source code to export the data. Appendix C details changes needed using the dummy module example.</p>	<p>The frame includes tools or functions to keep track of (i) single-value outputs (e.g., time of failure) per realization, (ii) dynamic outputs per realization (e.g., crack opening displacement as a function of time), and (iii) matrix outputs per realization, either static or dynamic (i.e., matrix output as a function of time).</p> <p>Recording an intermediate output is straightforward. Few lines of code or minimal changes are sufficient to record data, and the approach can be made systematic.</p> <p>No customized code or minimal customized code is needed to record data.</p>
<p>Effort for Graphical Display of Monte Carlo Outputs</p> <p></p> <p><u>Strengths:</u> Functions for graphic display are natively available in GoldSim and can be added with few steps.</p>	<p></p> <p><u>Strengths:</u> For the time series, the plotting functionality is automatically enabled for the recorded outputs, once the steps for recording multiple-realization output are implemented (see the previous entry for a brief discussion of these steps).</p>	<p>Making data available for plotting is straightforward. Few lines of code are sufficient to make data available for graphic display, and the approach to enable data for graphic display can be made systematic.</p> <p>Functions are readily available for the graphic display of data. The process to call graphic functions can be made systematic.</p>

Table 4-1. Evaluation of Frame Elements for the Development of Stochastic Models (Continued)		
xLPR-GoldSim	xLPR-SIAM	Expectation
Effort To Expose/Record Additional Outputs		
 <p>Strengths: Minimal steps are required to keep track of existing intermediate outputs (e.g., adding a graphical element into the model, and a dashboard shortcut). If the intermediate output of interest is tracked within a GoldSim submodel, a few extra steps are needed to pass the intermediate output to the root model.</p> <p>The reverse process to hide/remove intermediate outputs is straightforward. In general, removing elements is easier than adding elements. GoldSim models allow for full removal of containers and enclosed elements. Before running, GoldSim checks the model and warns the user of missing dependencies and their locations, facilitating the removal of extra elements.</p> <p>Limitations: For multidimensional (vector) data, exposing the data for subsequent export/display may require the addition of a large number of elements. Although groups of elements can be copied and renamed to reduce the workload, some settings must be manually adjusted for each GoldSim element. In a traditional programming language, repetitive tasks can be concisely programmed using indexing and loops. In GoldSim, much of the development is spent manipulating large numbers of GoldSim elements, with limited possibility to take advantage of programming shortcuts. GoldSim requires skill and patience (given the large number of elements that may require one-by-one modification) to implement changes. Appendix C provides details on changes needed to the xLPR code to expose additional output on the basis of the dummy module example.</p>	 <p>Limitations: Depending on the output, different steps are required to expose intermediate outputs. Exposing parameter output for post-processing (and plotting) requires additions in multiple source code files including defining export file names/interfaces, declaring arrays in the FORTRAN timeloop, and recompiling of the FORTRAN time loop file to be accessible to Python (see Appendix C). The majority of the effort, using the dummy module as an example, was spent identifying locations for additions and modifications to the source code. It is noted that the needed modifications to the xLPR source code were straightforward and did not require significant time to implement.</p> <p>The reverse process to hide/remove tracking of intermediate outputs requires the same number of steps. Shortcuts can be adopted (e.g., avoid creating the output file as last step); however, shortcuts do not fully clean the source code and may still track unnecessary data in memory.</p>	<p>A few steps are sufficient to enable tracking of existing output. A few lines of code are sufficient to track/exposed outputs.</p> <p>The approach to track/expose output can be made systematic.</p> <p>Tracked/exposed output is available to be extracted for post-processing and graphic display.</p> <p>The coding time needed to track/expose output is short.</p> <p>The process to track intermediate outputs can be easily reverted to hide/remove tracked outputs.</p>

Discussion and Recommendations

xLPR-GoldSim offers advantages to incorporate code updates and perform code maintenance. The time needed to become familiar with GoldSim is shorter, modifications can be focused, and modifications can be developed and tested independently from the main model. The unit-sensitive approach and active error checking (syntax and variable names) available in GoldSim minimize errors while programming. On the other hand, GoldSim requires skill and patience. As the number of elements in a model grows and interconnections develop, modifications need to be implemented element by element and the process can become cumbersome. This was experienced in xLPR-GoldSim while adding a new stochastic parameter as part of the dummy module insertion experiment. xLPR-SIAM requires familiarity with object-oriented programming to understand the code logic, as well as with Python, FORTRAN, and Python libraries. Systematic approaches were sought in SIAM (i.e., approaches that could be repeated to implement common actions during model development or maintenance, and to avoid the need of customized coding), and systematic approaches were found to define stochastic input parameters, data management of input parameters, and graphic display of input parameters. On the other hand, general systematic approaches for management of data output from multiple realizations appear difficult to define. The approach to capture/manage multiple-realization output data from a dummy module required changes to Python and FORTRAN codes. The changes needed would be different depending on which data and data structure (e.g., single valued, timestep series, static or timestep-dependent arrays) are to be captured. Given the need of customized coding, a higher level of expertise is needed to provide support to the xLPR-SIAM. One of the benefits of the xLPR-SIAM code is that once the required change was identified in the example of the insertion of a dummy module, a few lines of code spread among several modules sufficed to perform a broad set of functions (e.g., create output text files, enable text files for TRANSFORMERS and EXPECTATION, and enable the results from post-processing for graphic display, with all the graphic display options available). In contrast, in xLPR-GoldSim, every action needs to be manually implemented.

Recommendations

xLRP-GoldSim

The shortcomings noted in Table 4-1 for xLPR-GoldSim are mostly shortcomings of the GoldSim software. Those shortcomings can only be addressed by the GoldSim developers. No further recommendations are provided, other than enhancing the internal documentation, as explained in Chapter 3.

xLPR-SIAM

- Develop generic functions or approaches (i.e., systematic steps) to perform data management from multiple-realization outputs. Define standard data structures and data text formats for different kinds of outputs: single value outputs per realization, time series per realization, data arrays, and data arrays that are functions of time per realization. The purpose of the standard functions and data formats would be to minimize the need for a customized code to perform routine actions, such as exporting output data in text files or tracking intermediate output data in databases.
- Write tutorials for programmers. NRC expressed interest in actively contributing to the code development effort; tutorials would facilitate this effort. These tutorials should cover

procedures to (i) insert a new random variable, (ii) pass sampled values as input to a computational module, (iii) capture outputs in text files from multiple realizations, and (iv) define standard output file formats for various types of multiple-realization outputs (single value outputs per realization, time series per realization, data arrays, and data arrays that are functions of time per realization). Appendix C describes steps taken to implement the dummy module, which could be used as an example for developing a tutorial. Tutorials could help reduce time requirements for learning Python and other libraries used in the framework.

- Develop a list of optional intermediate outputs to be exported in text files. Use flags in the GUI to enable export of these intermediate outputs.
- At some point during the development of xLPR-SIAM, outputs were tracked in a database that could be queried using Python. It appears that such an approach was abandoned in favor of external text output files. It is recommended to reconsider maintaining a database, possibly tracking more intermediate outputs than in the text files. In this manner, the database could be queried within Python to access more data for further analysis or post-processing. A GUI could be developed with flags to enable/disable tracking of intermediate outputs in the database. If additional data are needed, Python functions could be developed to get data from the database and make text files. Alternatively, those data could be analyzed and plotted using Python, without the need for intermediate text files. It is envisioned that Python scripts could be developed by users and shared among users to perform particular analyses or get particular graphic displays or data post-processing. For example, Python post-processing scripts can be developed to perform the same functions as the TRANSFORMERS and EXPECTATION programs, or to parse, filter, and plot particular data.
- Design a strategy to allow for user-defined default input values for the deterministic cases.
- Consider developing a statistical toolbox to compute correlations between single-value inputs and single-value outputs per realization, accounting for the aleatory-epistemic realization split. Allow sorting of the correlation coefficients by magnitude to identify the parameters with strongest correlation. Consider correlations among “raw” data and rank-transformed data.

5 FUTURE DEVELOPMENT POTENTIAL

The objective of this section is to comment on the potential of the frames and the availability of special functions for future development. The following aspects were considered:

(i) preprocessing (e.g., specialized sampling, importance sampling), (ii) post-processing (e.g., statistics, sensitivity analyses, computations to account for leak detection and mitigation), (iii) optimization and parameter estimation, (iv) parallel and distributed processing, (v) multiparty code development, and (vi) availability of third-party modules. The last two points were considered because NRC staff expressed interest in actively contributing to code development, as well as opening code development to people in the industry. Similarly to previous sections, the findings are presented in the form of a table, Table 5-1, but no star rating grade was used.

Table 5-1. Comparison of Frame Features To Support Future Development	
xLPR-GoldSim	xLPR-SIAM
Preprocessing (e.g., Specialized Sampling, Importance Sampling)	
Pre-processing can be accomplished. The xLPR-GoldSim developers demonstrated this concept by incorporating the Discrete Probability Distributions (DPD) module for stochastic parameter sampling via DLL elements.	Pre-processing modules can be developed in Python or with an interface in C/C++ or FORTRAN and linked to the code. Pre-processing modules can be incorporated in a seamless manner with the option for interaction with a GUI.
Postprocessing (e.g., Statistics, Sensitivity Analyses, Computations To Account for Leak Detection and Mitigation)	
Once GoldSim [®] completes a run, there is no access to the internally stored data for further computations, except by specialized functions offered by GoldSim (e.g., computation of statistics of multirealization data, computation of correlations matrices of inputs and outputs). To address this limitation, external programs (TRANSFORMERS and EXPECTATION) were developed in the xLPR project to post-process data. Post-processed data cannot be brought back into the same GoldSim model to take advantage of plotting capabilities.	Post-processing modules can be developed in Python or with an interface in C/C++ or FORTRAN and linked to the code in a seamless manner, as was done for TRANSFORMERS and EXPECTATION. Inputs for post-processing input can be specified from a custom GUI interface. Results from post-processing can be graphically displayed within the frame.
Optimization and Parameter Estimation	
GoldSim has an internal tool for optimization. In the optimization tool, stochastic input parameters are adjusted to optimize an objective function. This optimization tool can be used for parameter estimation. GoldSim also includes a tool for sensitivity analysis to sweep the value of an input parameter across a range, while keeping others constant. Of interest to the project is adopting adaptive sampling strategies to sample infrequent failure modes and automatically adjusting weight factors to expedite convergence of statistics. It is envisioned that approaches where the output of the previous realizations is used as input to define the input parameters for the next realization following an optimization algorithm can be implemented with the use of DLLs. The only envisioned limitation is that execution of external optimization algorithms may be limited to the nonparallel run mode.	Adoption of sampling strategies where the output of previous realizations is used to define the inputs for the next realization (e.g., optimization for parameter estimation, adaptive sampling) could require changes to framework architecture, which would potentially require significant development effort. External software is readily available for optimization and parameter estimation (e.g., Parameter Estimation, Inc., 2011). Significant changes would be needed to allow for xLPR computations to interface with such external software.

Table 5-1. Comparison of Frame Features to Support Future Development (Continued)	
xLPR-GoldSim	xLPR-SIAM
Parallel and Distributed Processing	
<p>GoldSim already has embedded capabilities for parallel processing. Minimal changes are required to take advantage of parallel processing, except when DLLs with special requirements are used. The code reviewers ran multiple realizations of xLPR-GoldSim in parallel mode using four processors in a computer (see Chapter 2).</p> <p>Parallel runs cannot be made using the GoldSim Player. GoldSim Pro allows spanning realizations in up to four computer processors. The GoldSim Distributed Processing Module Plus allows for use of an unlimited number of computer processors. The license to operate the Plus module must be purchased separately.</p>	<p>There is flexibility for implementation of various parallel processing methodologies and technologies [e.g., Message Passing Interface (2011), Bulk Synchronous Parallel (Hill, et al., 1998), cloud computing]. There are several third party modules available that support object sharing and parallelization of Python-based code [e.g., Parallel Python (Vanovschi, 2011), Pyros (de Jone, 2011)].</p> <p>Making SIAM parallel will likely require significant development effort. It is recommended to implement changes to SIAM to enable parallel processing while development of xLPR is still in early stages.</p>
Multiparty Code Development	
<p>Limited capability is available in GoldSim for tracking changes to a model. The GoldSim version tracking function is limited to listing, in a text file, model elements affected by changes, without further details on those changes. There is no capability to revert individual changes.</p> <p>Because the model file is a proprietary format, external version tracking systems are not available. However, alternative effective approaches and protocols can be implemented. For example, file servers including intermediate model files can keep copies of most recent model files for download and upload after model changes. A custodian is needed for this approach to be effective. No major limitation is envisioned for multiparty code development.</p>	<p>There are established methods for tracking changes to Python source code, and there are several robust, open-source packages available for use [e.g., Mercurial (Mackall, 2011), SubVersion (Apache Software Foundation, 2011), CVS (Free Software Foundation, Inc., 2011), git (git, 2011), Bazaar (Canonical Ltd, 2011)].</p> <p>Changes to source code can be tracked by each individual change, and developers can revert to past states using code versioning tracking software. No major limitation is envisioned for multiparty development.</p>
Availability of Third-Party Modules	
<p>Third-party modules could potentially be incorporated through the use of external DLL elements. Third-party modules that cannot be translated into a DLL cannot be used directly in GoldSim.</p>	<p>There are many third-party modules available for a wide variety of applications (e.g., sampling, parallel processing, plotting, generation of documentation) developed for use with Python. Python libraries and modules are, in general, open source.</p> <p>No limitations are envisioned with respect to the use of third party modules designed for use with Python.</p>
<p>Apache Software Foundation. "Apache™ Subversion®." <http://subversion.apache.org/> Forest Hill, Maryland: The Apache Software Foundation. 2011.</p> <p>Canonical Ltd. "Bazaar." <http://bazaar.canonical.com/en/> London, United Kingdom: Canonical Ltd. 2011.</p> <p>Free Software Foundation, Inc. "CVS—Concurrent Versions System." <http://www.nongnu.org/cvs/#development> Boston, Massachusetts: Free Software Foundation, Inc. 2011.</p> <p>git. "The Fast Version Control System." <http://git-scm.com/> 2011.</p> <p>Hill, J., B. McColl, D. Stefanescu, M. Goudreau, K. Lang, S. Rao, T. Suel, T. Tsantilas, and R. Bisseling. "Parallel Computing." <i>BSPlib: The BSP Programming Library</i>. Vol. 24. pp. 1,947–1,980. 1998.</p> <p>de Jone, I. "Pyro—Python Remote Objects." <http://www.xs4all.nl/~irmen/pyro3/> 2011.</p> <p>Mackall, M. "Mercurial." <http://mercurial.selenic.com/> 2011.</p> <p>Message Passing Interface. "MPI Documents." <http://www.mpi-forum.org/docs/docs.html> 2011.</p> <p>Parameter Estimation, Inc. "PEST—Parameter Estimation for Any Model." <http://www.parameter-estimation.com/html/pest.html> Sandy, Utah: Parameter Estimation, Inc., a Division of Scientific Software Group. 2011.</p> <p>Vanovschi, V. "Parallel Python." <http://www.parallelpython.com/> 2011.</p>	

Recommendations

For xLPR-GoldSim, the main perceived limitation for future development is the lack of access to data stored in a GoldSim model file after run completion to allow for further analysis of the data without leaving the GoldSim frame. To overcome this limitation, GoldSim developers would have to enable additional functionality to open a window to manipulate archived data in the model files. It is unclear how difficult and expensive it would be for GoldSim developers to enable this functionality.

For xLPR-SIAM, it is recommended to implement changes to SIAM to take advantage of existing Python parallel processing capabilities. Such changes should take place early in the development of xLPR. Consideration should be given to changing the code architecture to allow for the use of external optimization software to drive the xLPR computations.

6 CONCLUSIONS

In general, xLPR-GoldSim and xLPR-SIAM have different limitations and strengths in regard to future development potential. xLPR-GoldSim is built upon GoldSim, which offers major strengths with respect to prompt model deployment, polished interfaces, graphic display, management of Monte Carlo data, the limited background needed to read GoldSim model files, and the quick learning time for model developers. GoldSim is a frame with numerous predefined functions that can be used in a “plug-play” approach. When specialized functions or approaches are needed, workarounds are possible to adapt existing GoldSim functions to perform different tasks. However, these workarounds, in general, compromise model clarity by the use of complex logic. Also, as GoldSim models grow in complexity, modifications can become cumbersome because adjustments need to be manually implemented in the fields of the GoldSim elements, element by element. In the comparison tables for Chapters 2 to 5, a number of limitations of xLPR-GoldSim were noted. However, the majority of those limitations are intrinsic to the GoldSim software. In other words, these limitations cannot be addressed by changing the xLPR-GoldSim model, but by changes in the GoldSim software. A major limitation of the GoldSim software, noted in Chapter 5, is that once GoldSim completes a run, only limited tools are available to perform computations on data stored in a model file. This limitation was overcome in the xLPR project by manual exporting of data stored in GoldSim model files. This process was inefficient; as indicated in Table 2.2, the export process can take a few hours. An alternative data export process could be explored by using DLLs to capture data and write realization data in external files after each realization is completed. To analyze the exported data, analytical and plotting software other than GoldSim is needed. These are likely limitations that the xLPR project will encounter when additional models for failure of other components of the piping cooling system are developed using GoldSim. Chances are high that separate models would have to be programmed, each generating “raw data,” and external tools or software would have to be used to analyze the GoldSim raw data to define a total system metric of risk. The process for a total system analysis is envisioned to be complex, and possibly only experts would be able to perform such total system analyses. Summarizing, models developed in GoldSim can be quickly deployed, are readable, and require moderate effort to initiate model maintenance. On the limiting side, developers must deal with GoldSim software constraints with workarounds as the model grows in size and complexity, the requirement for external tools to analyze the data seems unavoidable, and data exporting can be a potential bottleneck.

xLPR-SIAM is built upon SIAM-PFM. The SIAM platform is intended to include tools for probabilistic risk assessment and to be extensible to address different problems. Accordingly, the reviewers evaluated SIAM for functionality that they considered should be included in a frame used for the development of stochastic models. Functionality was sought for (i) sampling input parameters from distribution functions, (ii) managing input data from multiple realizations, (iii) managing output data (e.g., single value per realization, time series per realization, multivalued per realization, or multivalued per timestep), and (iv) plotting capabilities for inputs and outputs. To offer the intended frame adaptability, such functionality should be available in a systematic manner, so that by adapting blocks of code with minor changes (thus minimizing the need to write new, customized, code), the functionality could be applied to different inputs and outputs, and, of course, be available for the development of varied models. In Chapter 4, xLPR-SIAM scored high in areas (i) and (ii). It was found that systematic code, applied in the same or similar manner, can be used to perform the functions (i) and (ii). Using the dummy module described in Chapter 4, it was found that few lines of code sufficed to (1) create a new parameter in an input tab in the xLPR-SIAM GUI with all of the pulldown choices, (2) create text log files, and (3) enable graphic displays of probability density or cumulative probability plots. It is remarkable the number of actions that can be accomplished with a few lines of code, thanks

to the xLPR-SIAM design. However, xLPR-SIAM did not score high in task (iii). Using the dummy module from Chapter 4, it was found that changes were needed in several Python files and a FORTRAN code and in compilation of the FORTRAN code to capture a time series from multiple realizations in a text file. Based on the Chapter 4 evaluation, it is extrapolated that different code changes are needed to export data, depending on which data are to be exported. Given the need for code customization, a deep level of expertise in Python and FORTRAN is required to properly maintain a model, and an even deeper level of expertise is needed for model development. It is recommended in Chapter 4 to define standard data interfaces and standard data management functions applicable in a systematic manner to capture multiple realization data in text files or databases. It is also recommended to write tutorials for model developers. These standard functions and tutorials would help modelers to focus on the development computational algorithms representing physical processes. Modelers would use as-needed, predefined functions from SIAM libraries to build stochastic models. To make SIAM accessible to a range of programmers learning would focus on the library functions and, to a limited extent, Python. Currently, to implement a model following the xLPR-SIAM example, extensive knowledge of object-oriented programming, Python, FORTRAN, and Python libraries is needed. With respect to the aspect (iv), plotting capabilities, a mixed score was determined in Chapter 4 for xLPR-SIAM. A well-structured systematic approach was identified for plotting data output from the post-processing script EXPECTATION. In general, only few extra lines of source code were needed, after capturing a time series in an external text file, to enable the corresponding data to be available in pulldown menus in the SIAM GUI for the plots of outputs. The identified shortcomings under (iv) are related to the lack of options to control the appearance of plots and the lack of plots for single-value outputs for realization. Thus, effort is needed to develop the back end of the SIAM frame (i.e., data management of multiple realization outputs, and plotting of outputs) to make it a general frame for the construction of stochastic models, accessible to a range of programmers with diverse experience. In Chapter 5, it was recommended for SIAM to incorporate existing technologies for parallel processing, especially during early development of xLPR. In contrast to xLPR-GoldSim, all of the limitations pointed out in Chapters 2–5 for xLPR-SIAM can be addressed with extra coding effort. SIAM offers potential for scalability and development of an integrated unit for a total system risk analysis of piping cooling systems.

Thus, two future options are envisioned for xLPR. GoldSim offers convenience at the cost of workarounds and the need for external tools. SIAM offers flexibility, scalability potential, and the possibility to develop integrated units for total risk assessments; however, extra investment is needed to build the frame to make it accessible to a range of programmers (from NRC and the industry). To enhance the contrast of these two alternatives, the cost of use of GoldSim or SIAM for NRC staff within the next 5 years was estimated. The unit of “cost” selected was the time of use. A longer time would be associated with a frame that is more expensive to be used. A time estimate, as opposed to a dollar quantity, was preferred, because such information can be inferred more directly from the evaluation in Chapters 2 to 5. However, when a translation of money into time was needed, a conversion of \$250/hour was used.

The following assumptions were adopted to estimate the time of use.

- Assumption 1: SIAM is polished to allow programmers within the NRC staff to use it.
- Assumption 2: Every year, a module for failure of the cooling piping system is implemented.
- Assumption 3: Every year, one NRC staff member is trained on the use of the frame to ensure continuity (account for rotations and transfers).

Assumption 4: To translate time into money, the following equivalence was used: 1 hour = \$250.

Assumption 1, SIAM is further developed, is adopted for consistency with the expressed interest of NRC staff to actively participate in model development. As discussed in Chapter 4, to make SIAM accessible to a broader range of programmers, further development is recommended. If this development is not adopted, then time estimates for the use of the SIAM frame would have to be revised, as the time needed to attain enough familiarity would greatly increase.

Assumption 2, one module per year, conveys the point that xLPR is intended to move beyond the pilot problem study. Assumption 3, more staff are trained every year, is adopted to account for the possibility that NRC staff may go into rotation or be transferred to other divisions, and it would be safer to spread the responsibility for model development among various members of the NRC staff to ensure continuity and as a knowledge management strategy. Assumption 4, 1 hour = \$250, was set to translate money into time for a few factors that incur direct costs (e.g., cost of GoldSim licenses, cost of training by the GoldSim Technology Group LLC).

Time estimates were elicited from staff working on Chapters 2 to 5. The evaluators were asked to estimate the minimum and maximum time to undertake a task. To account for uncertainty, it was assumed that the actual task time could be any time between the minimum and maximum, and follow a uniform distribution. Thus, the average time, μ , and variance, σ^2 , were computed according to the equations

$$\mu = 0.5(a + b) \quad (6-1)$$

$$\sigma^2 = \frac{(b - a)^2}{12} \quad (6-2)$$

where a and b are the minimum and maximum time estimates. Time estimates for tasks for the use of GoldSim and SIAM are summarized in Tables 6-1, 6-2, and 6-3.

Tables 6-1 and 6-2 summarize estimates of the time it would take the NRC staff to use the GoldSim and SIAM frames. In the case of GoldSim (Table 6-1), it was assumed that GoldSim Technology Group LLC would provide yearly training. It was also assumed that NRC staff would use three GoldSim licenses to perform analyses and model development, and that staff would pay for GoldSim license maintenance fees to have access to recent versions of GoldSim. A conversion of \$250/hour was used to translate dollars into hours. In the case of SIAM (Table 6-2), the fact that SIAM libraries and dependencies are open source (i.e., are available at no monetary cost) was accounted for. The main difference in Tables 6-1 and 6-2 is in the time needed to launch runs and obtain data to perform analyses. For example, in the case of GoldSim, it was considered that it could take up to 6 hours per run (modeler time, not runtime) for a modeler to set the run, organize the data, and execute post-processing to derive meaningful results. In the case of SIAM, it was considered that it could take a maximum of 1 hour, assuming the SIAM frame is developed to a mature state. On average, using information in the Totals row in Tables 6-1 and 6-2, it is concluded that NRC staff would spend less time using the SIAM frame, assuming SIAM is developed to a more mature state.

In computing the totals in the last row in Tables 6-1 and 6-2 some costs were assumed to be incurred only once in a 5-year period (e.g., GoldSim licenses are purchased only once; post-processing scripts to analyze GoldSim data are programmed only in the first year). These tasks are labeled with the phrase “one-time cost” in Tables 6-1 and 6-2. All other tasks not

Table 6-1. Estimate of the Time for GoldSim Use by NRC Staff						
Task	Min (hrs)	Max (hrs)	Mean (hrs)	Variance (hr ²)	S. Dev. (hrs)	
Development of New Model for Failure of Component of Piping Cooling System (Per Year)						
Implementation of new model into frame, assuming mathematical algorithms are mature	200	400	300	3,333.33	57.73	
Documentation to execute the model	120	160	140	133.33	11.55	
Training (Per Year)						
Hiring GoldSim Technology Group LLC for 4-day training: \$10,500 + travel expenses (\$2,000)	50	50	50	0.00	0.00	
NRC staff programmer learning GoldSim, FORTRAN basics, and DLL interface	40	120	80	533.33	23.09	
Code Use						
Obtain data to perform analyses; 40 run executions in a year (per year)	40	240	140	3,333.33	57.73	
Develop additional scripts to analyze data (e.g., use Mathematica, Matlab, Excel, or other external tool to perform computations). One-time cost.	40	200	120	2,133.33	46.19	
Develop additional scripts to compute correlations. One-time cost.	40	200	120	2,133.33	46.19	
Code Maintenance (Per Year)						
Emigrate codes to newer versions of GoldSim® and supporting libraries—only if effort is reasonable	40	160	100	1,200.00	34.64	
Testing of emigrated versions	40	160	100	1,200.00	34.64	
License Cost						
License maintenance fee (per year). Assume three licenses are maintained. \$1,600 per license.	19.2	19.2	19.2	0.00	0.00	
3 GoldSim licenses: \$3,950 per license. One-time cost.	47.4	47.4	47.4	0.00	0.00	
Totals						
Total Time During 5-Year Use			4,933.40	52,933.26	225.98	

Table 6-2. Estimate of the Time for SIAM Use by NRC Staff					
Task	Min (hrs)	Max (hrs)	Mean (hrs)	Variance (hr ²)	S. Dev. (hrs)
Development of New Model for Failure of Component of Piping Cooling System (Per Year)					
Implementation of new model into frame, assuming mathematical algorithms are mature	200	400	300	3,333.33	57.73
Documentation to execute the model	80	160	120	533.33	23.09
Training (Per Year)					
Training NRC staff on use of SIAM frame for development of stochastic models	40	80	60	133.33	11.55
Training NRC staff programmer on Python, FORTRAN, and Python libraries	80	180	130	833.33	28.87
Code Use (Per Year)					
Obtain data to perform analyses; 40 run executions in a year	20	40	30	33.33	5.77
Develop additional scripts to analyze data	0	0	0.0	0.00	0.00
Develop additional scripts to compute correlations	0	0	0.0	0.00	0.00
Code Maintenance (Per Year)					
Emigrate codes to newer versions of Python and supporting libraries—only if effort is reasonable	40	160	100	1,200.00	34.64
Testing of emigrated versions	40	320	180	6,533.33	80.83
License Cost					
License maintenance fee	0	0	0.0	0.00	0.00
Cost per license	0	0	0.0	0.00	0.00
Totals					
Total Time During 5-Year Use			4,600.00	62,999.90	251.00

Table 6-3. Estimate of Time To Finalize the SIAM Frame					
Task	Min (hrs)	Max (hrs)	Mean (hrs)	Variance (hr ²)	S. Dev. (hrs)
Make data management of outputs standard and systematic. See recommendations in Chapter 4.	240	600	420	10,800	103.92
Enhance plotting capabilities (e.g., allow for user control of plotting options, plots of single-value outputs per realization, horse-tail plots). See recommendations in Chapter 2.	160	320	240	2,133.33	46.19
Allow for deterministic runs with user-defined inputs or stochastic variables fixed at a quantile or mean values. See recommendations in Chapter 2.	40	160	100	1,200.00	34.64
Parallelize xLPR. See recommendations in Chapter 5.	200	480	340	6,533.33	80.83
Compute correlations between single-value inputs and single-value outputs (considering epistemic-aleatory realization split). Sort parameters by correlation value to support sensitivity analyses. Consider rank correlations and raw data correlations. See recommendations in Chapter 4.	80	320	200	4,800.00	69.28
Modify SIAM to facilitate the use of readily available external optimization or parameter estimation tools. See recommendations in Chapter 5.	80	240	160	2,133.33	46.19
Document SIAM for programmers (users of the SIAM frame to develop other models). Enhance internal documentation by consistent use of docstrings in Python objects. Develop tutorials for (i) adding stochastic input parameters, (ii) managing multiple-realization outputs (e.g., single value, time series, matrix outputs), and (iii) graphic display of multiple-realization inputs and outputs (e.g., PDF, CDF, output versus time). Develop tutorials for the consistent use of parallelization functions. Provide examples on how to interface with GUIs. See recommendations in Chapters 3, 4, 5.	240	560	400	8,533.33	92.38
Totals					
Total Development Time			1,860.00	36,133.32	190.09
Total Time Including Totals from Table 6.2			6,460.00	99,133.22	314.854

including the “one-time cost” label are assumed performed every year. The total average and variance in the last row in Tables 6-1 and 6-2, assumed that some tasks are performed once and other tasks are performed every year in the 5-year period. The total mean and total variance are the sums of the means and variances of the tasks performed in the 5-year period.

The time to develop SIAM to a mature state needs to be considered as does the need to obtain total estimates for the cost of the SIAM frame. Table 6-3 provides estimates of the time needed for SIAM developers to complete the frame. Recommendations provided in Chapters 2 to 5 were used in defining the entries in Table 6-3. The total mean time and total variance are included in the last row in Table 6-3 (including totals from Table 6-2). When the SIAM development time is taken into account, the total time associated with the use of the SIAM frame exceeds the time associated with the use of the GoldSim frame by approximately 30 percent.

Figure 6-1 shows approximate probability distributions for the time associated with the use of the GoldSim and SIAM frames in a 5-year period. For SIAM, the distributions include the upfront development needed to complete the development of the frame. The distributions are assumed to be normal (a reasonable assumption given the central limit theorem and the large number of entries in the sum to compute total times), with means and standard deviations as in the last rows in Tables 6-1 and 6-3.

Conclusion

It is concluded that use of the SIAM frame over a 5-year period would be more expensive than use of the GoldSim frame, under a set of defined assumptions. On average, SIAM was estimated to be approximately 30 percent more expensive. The estimated range of costs, expressed in time, is presented in Figure 6.1. On the other hand, it is estimated that NRC staff would spend less time using the SIAM frame (provided SIAM is developed to a more mature state) than using the GoldSim frame. SIAM would be expected to be developed to a stage such that models would automatically incorporate tools for post-processing, making the use of SIAM more convenient. In contrast, significant user intervention is expected in GoldSim models to analyze and interpret output data, unless GoldSim Technology Group LLC develops an approach to access and manipulate data stored in model files that does not require exporting to external text files (and addresses other limitations noted in Chapters 2 to 5).

Therefore, NRC staff may opt to spend more to sponsor the development and use of SIAM to gain flexibility and convenience. Appropriate consideration should be given to the risk and cost of software development (especially in the absence of a commercial entity committed to long-term support and software maintenance) and frequent change in hardware, operating systems, and third-party software.

NRC staff may opt to use GoldSim and save some resources, at the cost of more user intervention, to execute models and analyze output data. NRC staff may request that GoldSim Technology Group LLC enhance flexibility in model development and enhance access to data stored in model files. It is unclear how difficult it would be for the GoldSim Technology Group LLC to address the shortcomings noted in Chapters 2 to 5. Contact with GoldSim Technology Group LLC is recommended to determine whether those shortcomings could be addressed in future versions of GoldSim.

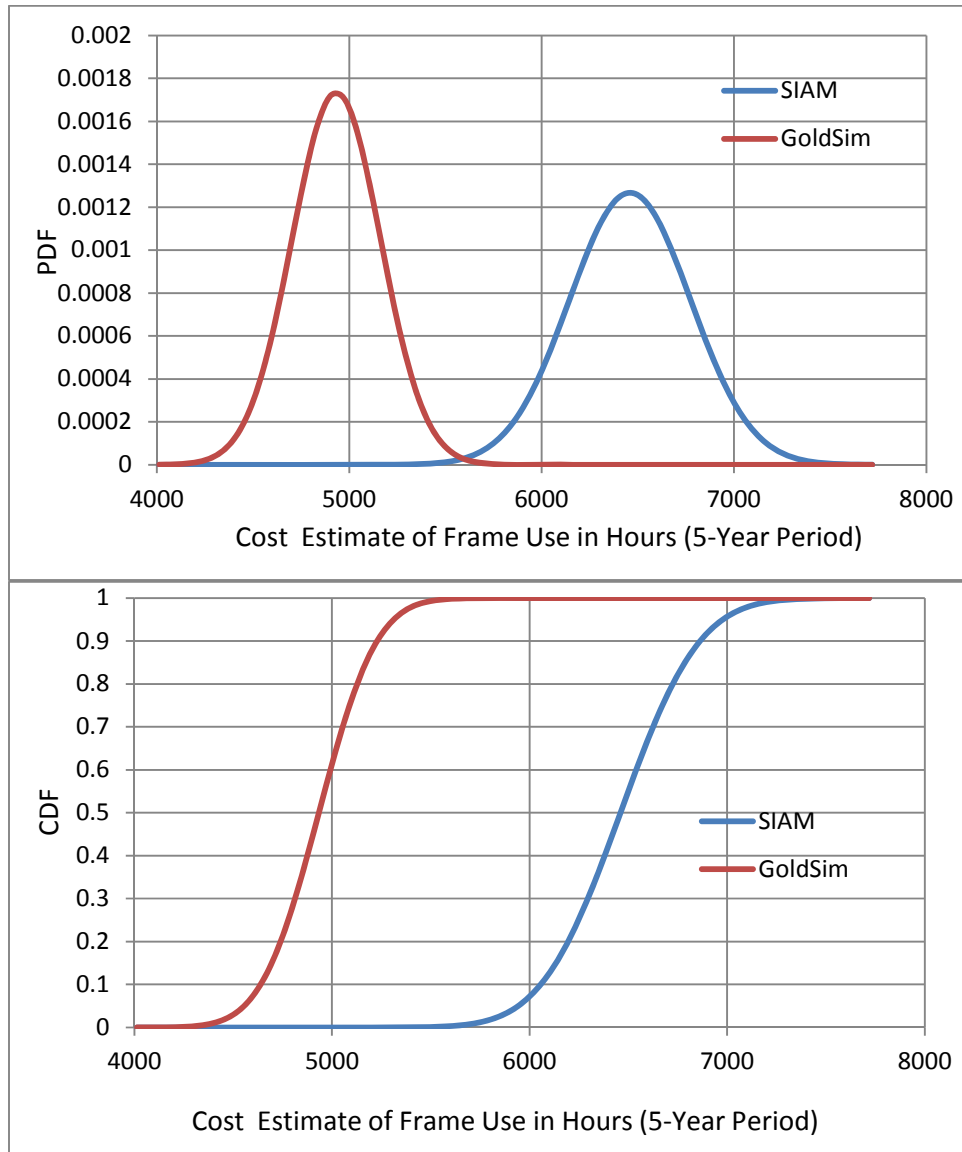


Figure 6-1. Time Estimates for the Cost of Using the GoldSim and SIAM Frames in a 5-Year Period, Expressed as Probability Distribution and Cumulative Distribution Functions

7 REFERENCES

(Some open software does not have a publisher or specific author. In that case, the name of the software and website is provided as reference.)

Adams, B.M., W.J. Bohnhoff, K.R. Dalbey, J.P. Eddy, M.S. Eldred, D.M. Gay, K. Haskell, P.D. Hough, and L.P. Swiler. “DAKOTA, A Multilevel Parallel Object-Oriented Framework for Design Optimization, Parameter Estimation, Uncertainty Quantification, and Sensitivity Analysis, Version 5.1 User’s Manual.” SAND2010–2183. Albuquerque, New Mexico: Sandia National Laboratories. 2010.

Brandl, G. “Sphinx—Python Documentation Generator.” 2011. <<http://sphinx.pocoo.org/>>

GoldSim Technology Group LLC. “GoldSim, Version 10.11.” <www.GoldSim.com> Issaquah, Washington: GoldSim Technology Group. 2011.

Klasky, H.B., P.T. Williams, B.R. Bass, and S. Yin. “SIAM-xLPR Version 1.0 Framework Report.” ORNL/NRC/LTR–248. Oak Ridge, Tennessee: Oak Ridge National Laboratory. 2010.

Loper, E. “Epydoc—Automatic API Documentation Generation for Python.” 2011. <<http://epydoc.sourceforge.net/>>

Mattie, P.D., C.J. Sallaberry, J.C. Helton, and D.A. Kalinich. “Development, Analysis, and Evaluation of a Commercial Software Framework for the Study of Extremely Low Probability of Rupture (xLPR) Events at Nuclear Power Plants.” SAND2010–8480. Albuquerque, New Mexico: Sandia National Laboratories. 2010.

NRC and EPRI. “xLPR Version 1.0 Report, Technical Basis and Pilot Study Problem Results.” Washington, DC: NRC, Office of Nuclear Regulatory Research and Palo Alto, California: EPRI. 2011.

Python Software Foundation. “pydoc—Documentation Generator and Online Help System.” *Python Programming Language—Official Website*. 2011a. <<http://docs.python.org/release/2.6.6/library/pydoc.html?highlight=pydoc#module-pydoc>>

Python Software Foundation. “doctest—Test interactive Python examples.” *Python Programming Language—Official Website*. 2011b. <<http://docs.python.org/release/2.6.6/library/doctest.html?highlight=doctest#module-doctest>>

Python Software Foundation. “Built-in Functions.” *Python Programming Language—Official Website*. 2011c. <<http://docs.python.org/release/2.6.6/library/functions.html>>

Rathmann, U. “Qwt—Qt Widgets for Technical Applications.” 2011. <<http://qwt.sourceforge.net/>>

Riverbank Computing Limited. “PyQt4.” <<http://www.riverbankcomputing.co.uk/software/pyqt/intro>> Wimborne, Dorset, United Kingdom: Riverbank Computing Limited. 2011.

Swiler, L.P. and G.D. Wyss. "A User's Guide to Sandia's Latin Hypercube Sampling Software: LHS Unix Library/Standalone Version." SAND2004-2439. Albuquerque, New Mexico: Sandia National Laboratories. 2004.

Vermeulen, G. "PyQwt Plots Data With Numerical Python and PyQt." 2011.
<<http://pyqwt.sourceforge.net/>>

APPENDIX A

A1 SOFTWARE QUALITY ASSURANCE

A1.1 Background

10 CFR Part 50, Appendix B identifies a number of requirements applicable to software used in safety-related applications. Criterion II states, in part, that activities affecting quality must be accomplished under suitably controlled conditions and that the quality assurance program shall provide control over activities affecting the quality of the identified structures, systems, and components, to an extent consistent with their importance to safety. Controlled conditions include use of appropriate equipment, suitable environmental conditions for accomplishing the activity, and assurance that all prerequisites have been satisfied. Criterion III requires, in part, that design control measures provide for verifying or checking the adequacy of the design. Essentially, 10 CFR Part 50, Appendix B requires that software used in safety-related applications must be controlled to an extent consistent with its importance to safety.

U.S. Nuclear Regulatory Commission (NRC) has endorsed industry guidance relating to quality assurance in Regulatory Guide 1.28 (NRC, 2010) in ASME NQA-1a-2009 Addenda (ASME, 2009). NQA-1, Requirement 3, Section 401 states in part that

- (a) “The computer program shall be verified to show that it produces correct solutions for the encoded mathematical model within the limits for each parameter employed,
- (b) The encoded mathematical model shall be shown to produce a valid solution to the physical problem associated with the particular application.”

Further requirements are provided in NQA-1, Subpart 2.7 for “the acquisition, development, operation, maintenance, and retirement of software.” NQA-1 Subpart 2.7 follows the common software quality assurance industry approach of identifying formal quality assurance controls associated with each of the software development life cycle phases: (i) design requirements definition, (ii) design, (iii) implementation, (iv) test, and sometimes (v) installation. For the software life cycle phases, requirements are identified for documentation, review, software configuration management (CM), and problem reporting and resolution.

For software to be used in safety-related applications, the software quality assurance program must establish and implement controls to an extent necessary to demonstrate that the computer program produces correct solutions for the encoded mathematical model within the limits for each parameter employed (generally described as software verification and validation) and the encoded mathematical model produces a valid solution to the physical problem associated with the particular application (generally described as model validation).

A2 SOFTWARE QUALITY ASSURANCE FOR EXTREMELY LOW PROBABILITY OF RUPTURE

A2.1 Extremely Low Probability of Rupture Project Pilot Study Quality Assurance

The approach to quality assurance during the extremely low probability of rupture (xLPR) pilot study is described in (Sandia National Laboratories, 2009). The CM plan addresses the xLPR program commitment to implement the four key principles of CM as follows:

- Configuration Identification: Identify and name each major piece of the model. In the remainder of this plan, these major pieces of the model are referred to as configuration items (CIs).
- Configuration Control: Control changes to the CIs.
- Status Accounting: Track the status of the CIs.
- Verification and Audit: Confirm that the previous three principles are being implemented correctly.

The CM further describes the verification and audit as including

- Model checking
- Software qualification checking
- Independent verification and validation
- Model file set verification
- Software verification checking of modules
- Data verification (input values)
- Model input independent verification and validation
- Model output graphic/plot checking

The two program participants (Sandia National Laboratories and Oak Ridge National Laboratory) developed and employed documented desktop instructions and standard forms and templates (including checklists) that facilitated implementation of the CM plan. Desktop instructions addressed

- Programming practices
- Model development CM
- Model parameters
- Model changes
- Model documentation and checking
- Model output plot preparation

The participants also employed the software tools Microsoft® SharePoint® (Microsoft Corporation, 2010) and Mercurial (O'Sullivan, 2009) for CM and to facilitate collaborative model and code development.

The pilot study quality assurance practices centered on CM and checking and did not attempt to implement the full range of NQA-1, Subpart 2.7 software quality assurance requirements. The focus on CM and checking is appropriate considering the prototyping nature of the pilot study and the expectation of full scope, formal quality assurance controls in subsequent development projects. In regard to model validation, the pilot study included preparation and execution of several deterministic test problems.

A2.2 Software Quality Assurance for Future Extremely Low Probability of Rupture Development

Preparation of software quality assurance plans addressing the full range of NQA-1, Subpart 2.7 requirements should be the first step in subsequent development of xLPR software. Considering the complexity and uncertainties in the mathematical models employed in the xLPR software, the greatest challenge may be in defining model validation “to an extent consistent with its importance to safety.” In the face of large uncertainties, an expert panel of peer reviewers, such as described in NUREG-1297 (NRC, 1988) may provide additional confidence for model validation. NUREG-1297 suggests that “a peer review should be used when the adequacy of information (e.g., data, interpretations, test results, design assumptions, etc.) or the suitability of procedures and methods essential to showing that the ... system meets or exceeds performance requirements ... [that] cannot otherwise be established through testing, alternate calculations, or reference to previously established standards and practices.” Alternatively, an expert elicitation such as that described in NUREG-1563 (NRC, 1996) could be employed to obtain expert judgments in a formal, highly structured approach that may provide greater confidence in the model validation approach. Thorough documentation and reporting of peer reviews and expert elicitations may be used in shareholder interactions to facilitate acceptance of the xLPR model as a valid design verification tool.

Other considerations for model validation should include the following.

- Model development and validation should be integrated in the requirements definition life cycle phase so that the mathematical model is as accurate as possible prior to starting the software design phase.
- Model development and validation should consider all potential sources of uncertainty in the physical system being represented and in the mathematical model.
- Stringent change controls should be applied if the mathematical model is refined after the requirements definition life cycle phase to ensure that changed requirements are properly implemented throughout.
- Validation testing should include comparison of model results with field data and laboratory experiments whenever possible.

Considerations for other areas of software quality assurance should include the following.

- The xLPR Project Pilot Study followed several good practices that should be carried on in future xLPR development, including use of documented desktop instructions and forms and templates.

- CM tool selection for use in subsequent xLPR development should be based on needs for developing a finished software product, which can be much different from the needs for prototype development.

A2.3 Summary

NQA–1, Subpart 2.7 provides appropriate software quality assurance requirements for future xLPR software development. Robust model validation is an essential quality assurance element for software intended for safety-related design analyses. Peer reviews and expert elicitations supporting model validation activities can be used to develop the necessary confidence in the xLPR mathematical models in their functions as producing valid solutions to the physical problem associated with the particular application.

A3 EXTREMELY LOW PROBABILITY OF RUPTURE MODEL VALIDATION

A3.1 Introduction

The purpose of model validation is to evaluate whether the deterministic, physical models coded into the computer program xLPR accurately predict the physical behavior for which they are intended. Two possible types of validation could be performed:

1. Experimental validation based on using measured full-scale (e.g., pipe) or small-scale (e.g., laboratory coupon) test data. Although full-scale tests can be very costly, there are frequently problems of transferring small-scale test data to validate physical models applied to full-scale components.
2. Analytical validation based on using accepted analytical tools (e.g., finite element codes) as accurate predictions of experimental behavior (e.g., displacement response of pipe to applied loads) and physical quantities [e.g., crack tip driving forces such as the J-Integral (J) and Crack Opening Displacement (COD)].

These two forms of validation testing are consistent with the quality assurance program in Subsection A2.2 (i.e., testing should include comparison of model results with laboratory experiments).

A3.2 Validation Topics

The following approaches would be used to validate modeling concepts and deterministic methods used in a number of modules contained in the xLPR code. The approaches presented in this section serve to provide a robust model validation, which, as discussed in Subsection A2.2, is considered an essential part of the quality assurance program for software intended for safety-related design analyses.

A3.2.1 Geometrical Modeling Issues: Representing a Surge Nozzle by a Pipe

Currently, xLPR uses the General Electric/Electric Power Research Institute (GE/EPRI) method for determining the necessary fracture parameters and the LBB.ENG2 method for elastic-plastic

fracture mechanics. For simplicity, these methods are based upon the geometry of a pipe. The analytical equations are expressed in terms of various wall thicknesses and radii. However, it would be useful to evaluate how well the current analytical models in xLPR based on relatively simple pipe geometries compare with the behavior of the geometrically more complex surge nozzles.

Therefore, validation would identify under what circumstances the approximate geometry modeling used in xLPR is conservative or nonconservative. The validation would involve comparing finite element results for various fracture parameters used in critical crack and leak-before-break assessments using both cracked surge nozzle and pipe geometries.

Specific topics to be addressed in this validation are outlined next.

A3.2.2 Through-Wall Cracks

The validation exercise would involve using finite element analyses to calculate J and measure COD and crack areas of through-wall cracks in dissimilar welds in a surge nozzle and a pipe under bending, axial force, and internal pressure and comparing these results with xLPR LBB.ENG2 predictions. The validation would be performed using finite element analyses because performing experiments could be problematic (e.g., the high cost of fabricating full-scale surge nozzles including welding and technical difficulties such as the need to fabricate sharp through-wall cracks).

A3.2.3 Surface Cracks

Currently, xLPR employs semi-elliptical surface crack limit load solutions for cracks in pipes based upon the work of Rahmann (1998). For generality, xLPR should be able to determine the surface crack stability failure condition under conditions that could promote failures before the limit load is reached. EPRI (2011) acknowledges this lack of generality and also states that an elastic-plastic fracture-mechanics-based approach will be required. However, no specific methodology for the elastic-plastic fracture mechanics calculations is currently proposed (EPRI, 2011).

Therefore, for different types of surface cracks, plastic collapse loads would be determined using finite element models of surge nozzles and pipes subjected to bending, axial force, and pressure loads. The finite-element-derived plastic collapse loads would be compared to the values predicted using the pipe solutions in xLPR to assess their accuracy.

A3.2.4 Load Modeling Issues: Representing Pressure Effects as an Axial Force

The xLPR code allows for the effects of internal pressure on fracture mechanics parameters by applying an axial load equal to the axial load exerted by the pressure on the ends of a closed pipe and the extra axial force that arises by the pressure acting on the faces of internal surface cracks. This approach is adequate for calculating linear elastic fracture parameters for circumferential cracks, but it has serious limitations for cracks under elastic-plastic and plastic collapse conditions. The most serious limitation is that plastic yielding in pressurized pipes is dominated by the hoop stress induced by the pressure, and the current approach in xLPR ignores this. (Of lesser concern is the effect of the moment produced by the pressure acting on the crack faces.) Hoop stress may be important when the pressure is high or the pipe is subject to high temperature, (e.g., this may lead to creep effects) (Kim, et al., 2002). Therefore, it is

considered important that the calculation of J and COD estimates for through-wall cracks includes the internal pressure component (Kim, et al., 2002). The results of Lei and Budden (2004) illustrate the important role the hoop stress plays on limit load predictions. Therefore, it is important that the effects of the hoop stress arising from internal pressure on the elastic-plastic fracture mechanics parameters xLPR employs (such as J, COD, and limit load solutions) be determined for through-wall and surface cracks to assess potential nonconservative elements in that code. This validation exercise can be efficiently accomplished by combining the finite element investigation of geometrical modeling issues described in Subsection A3.2.1 with a parallel finite element investigation of pressure effects. The finite element models built as part of the Subsection A3.2.1 work can be utilized in the finite element investigations described in this section.

A3.2.5 Material Modeling Issues: Representing Dissimilar Weld Stress-Strain Behavior by the Ramberg-Osgood Equation

A3.2.5.1 Ramberg-Osgood Equation

An approach for performing simple J and COD estimation calculations is the well-known GE/EPRI method. The GE/EPRI model is based on the Ramberg-Osgood material model in which the coefficient and strain hardening index is used as input for representing the material's stress-strain behavior. However, there is a potential problem in using the Ramberg-Osgood model and limiting J and COD analyses to this type of stress-strain behavior. Specifically, the problem is the ability of the Ramberg-Osgood model to accurately capture the measured stress-strain behavior (Brust, 1987; Gilles and Brust, 1991). Kim, et al. (2004), Chattopadhyay (2006), and Huh, et al. (2006) discuss how the J estimates are sensitive to the accuracy of data fitting using the Ramberg-Osgood model.

Currently, the xLPR developers do not appear to address uncertainties related to representing stress-strain behavior by Ramberg-Osgood equations.

Therefore, it would be appropriate to validate the accuracy of xLPR results because they depend on fitting Ramberg-Osgood stress-strain curves to the stress-strain data. Validation would involve evaluating finite element predictions of key parameters (e.g., J, crack areas) using actual measured stress-strain curves as input and comparing these results with the corresponding predictions of xLPR. Finite element models built as part of the work performed in Subsection A3.2.1 would be utilized in the finite element calculations outlined in this section.

A3.2.5.2 Dissimilar Welds

The current approach used in xLPR to assess cracked dissimilar welds is to use the stress-strain behavior of the base metal and the fracture properties of the weld metal. Support for this approach is cited based on comparison of predicted and experimentally measured fracture behaviors. However, this justification is not substantiated through analysis of cracks in the welds of surge nozzles. It is proposed that the issue of dissimilar metal welds [i.e., what is the most appropriate stress-strain curve to use (the weld or base or a combination of the two)] be investigated through finite element analysis of cracks in the welds of surge nozzles and pipes. This activity would be performed in conjunction with the finite-element-based investigations in Subsection A3.2.1.

A3.2.6 Crack Growth Rate Modeling Issues

The current crack growth rate model in xLPR is the MRP–115 model (EPRI, 2004). This model is a function of temperature and the stress intensity factor (K). Investigation of primary water stress corrosion cracking (PWSCC) crack growth is given in White, et al. (2008) and MRP–216 (EPRI, 2007), which include welding residual stresses. The methodology of White, et al. (2008) and MRP–216 (EPRI, 2007) is based on the use of linear elastic fracture mechanics to calculate K used in the MRP–115 model.

There are three issues related to crack growth rate modeling in xLPR that are of concern. First, as pointed out in EPRI (2011), there are limitations regarding the use of K (which is based on linear elastic fracture mechanics) to characterize stress corrosion cracking (SCC) rates in situations involving significant crack tip plasticity. The most important source of stress driving SCC in surge nozzles is welding residual stresses. The presence of high residual stresses may result in plasticity that could affect the validity of using the K for calculating crack growth rates (EPRI, 2011). Second, if crack tip plasticity effects are important, then the question arises as to what fracture mechanics parameter should be used to characterize SCC rates under these conditions. Third, after initiating, SCCs are small. It is known that the propagation rates of small cracks under cyclic loading conditions tend to be underpredicted by fatigue crack propagation rates measured on coupons containing relatively long cracks when the rates are characterized in terms of cyclic K . Although SCC occurs under static, as opposed to cyclic, loading conditions, the possibility that observations regarding fatigue crack growth rates of small cracks being larger than those of long cracks under similar crack tip loading conditions may also carry over to SCC of small and long cracks and needs to be addressed.

A3.2.7 Crack Tip Plasticity

The use of K can result in an underestimation of SCC rates if this parameter is used to characterize SCC rates when loading conditions cause significant crack tip plasticity. Under these conditions, J is the more appropriate fracture mechanics parameter to use. Finite element analysis of cracked surge nozzles in the presence of simulated welding residual stresses can be performed to establish whether crack tip plasticity effects are important. Typical residual stresses would be generated in the finite element models based on treating these stresses as initial strains, similarly to how thermal stresses are represented by thermal strains resulting from a temperature field. The through-wall residual stress variations presented in xLPR documents would be used to guide the choice of residual stresses to be used in this activity.

Solutions for J would be computed for cracks in linear elastic and elastic-plastic materials under similar loading conditions, and the results would be compared to assess the effects of plasticity on the crack tip driving force. Because SCC models are nonlinear functions of K , the J values can be converted to equivalent (plastically corrected) K values and computational simulations of crack growth in welded surge nozzles would be performed to establish the influence of crack tip plasticity on SCC lifetimes. Validation of the xLPR approach of using K can be accomplished if the difference in the two sets of predicted lifetimes (one set based on K and the other on a plastically corrected K) is acceptably small. If the result of this validation is not acceptable, then this activity could provide data to assist in modifying the linear elastic approach adopted in xLPR.

A3.2.8 Characterization of Stress Corrosion Cracking Rates When K Is Not Valid

If the results of the validation exercise performed in Subsection A3.2.7 demonstrate that K is not the appropriate parameter to use for calculating growth rates for cracks in the welds of surge nozzles due to the effects of high residual stresses, then it would be useful to experimentally compare SCC rates under linear elastic conditions with rates measured under elastic-plastic conditions to establish that J is the appropriate fracture parameter to characterize PWSCC rates. These tests need not involve residual stresses, because the intent would be to establish the principle that J (or an equivalent plastically corrected K) enables SCC rates to be extrapolated from the linear elastic regime (where J is directly related to K^2) into the elastic-plastic regime. Substantiation of this extrapolation would validate the use of J as a characterizing parameter for SCC.

A3.2.9 Small Cracks

Another important aspect of SCC crack growth is the effect of small cracks because any small crack enhancements to the SCC rates based on measurements made on long cracks will make xLPR predictions nonconservative.

One potential small crack problem is that when a crack is small, the crack tip plastic zone may be of the same size as the crack, which would make linear elastic fracture mechanics invalid (Miller, 1982; Smith, 1977; Anderson, 2005). This issue, the replacement of K by an elastic-plastic fracture mechanics parameters such as J when crack tip plasticity is significant, is similar to the validation issue addressed in Subsection A3.2.7 and would be resolved by the work proposed in that section.

Another potential small crack problem is that under environmentally assisted cracking conditions, small crack effects on growth rates may be occurring that are not related to crack tip plasticity issues, such as possible changes between long crack and small crack growth behavior due to differences in electrochemical reactions at the tips of these two kinds of cracks (Anderson, 2005). To resolve this issue, small-scale coupon tests would be used to validate the applicability of the PWSCC equations used in xLPR in the small crack regime. These tests would be performed under loading conditions that induce crack tip driving forces that are typical of the values likely to be encountered by small cracks in the welds of surge nozzles. The test environment (temperature and water chemistry) would be similar to that experienced by small cracks at surge nozzles.

A4 References

Anderson, T.L. *Fracture Mechanics: Fundamentals and Applications*. Boca Raton, Florida: CRC Press. 2005.

ASME. "Quality Assurance Requirements for Computer Software for Nuclear Facility Applications." ASME NQA-1a-2009 Addenda to ASME NQA-1-2008 Edition. New York City, New York: American Society of Mechanical Engineers. 2009.

Brust, F.W. NUREG/CR-4853, BMI-2145, "Approximate Methods for Fracture Analysis of Through-Wall Cracked Pipes." Washington, DC: NRC, Office of Nuclear Regulatory Research. 1987.

Chattopadhyay, J. "Improved J and COD Estimation by GE/EPRI Method in Elastic to Fully Plastic Transition Zone." *Engineering Fracture Mechanics*. Vol. 73. pp. 1,959–1,979. 2006.

EPRI. "Materials Reliability Program: Models and Inputs Selected for Use in the xLPR Pilot Study (MRP–302)." Palo Alto, California: EPRI. 2011.

EPRI. "Material Reliability Program: Advanced FEA Evaluation of Growth and Postulated Circumferential PWSCC Flaws in Pressurizer Nozzle Dissimilar Metal Welds (MRP–216)." Palo Alto, California: EPRI. 2007.

EPRI. "Materials Reliability Program: Crack Growth Rates for Evaluating Primary Water Stress Corrosion Cracking (PWSCC) of Alloy 82, 182, and 132 Welds (MRP–115)." Palo Alto, California: EPRI. 2004.

Gilles, P. and F.W. Brust. "Approximate Fracture Methods for Pipes—Part I: Theory." *Nuclear Engineering and Design*. Vol. 127. pp. 1–17. 1991.

Huh, N.-S., Y.-J. Kim, and K.-B. Yoon. "Influence of Ramberg-Osgood Fitting on the Determination of Plastic Displacement Rates in Creep Crack Growth Testing." *Fatigue and Fracture of Engineering Materials and Structures*. Vol. 30. pp. 277–286. 2006.

Kim, Y.-J., N.-S. Huh, Y.-J. Kim, Y.-H. Choi, and J.-S. Yang. "On Relevant Ramberg-Osgood Fit to Engineering Nonlinear Fracture Mechanics Analysis." *Journal of Pressure Vessel Technology*. Vol. 126. pp. 277–283. 2004.

Kim, Y.-J., N.-S. Huh, and Y.-J. Kim. "Quantification of Pressure-Induced Hoop Stress Effect on Fracture Analysis of Circumferential Through-Wall Cracked Pipes." *Engineering Fracture Mechanics*. Vol. 69. pp. 1,249–1,267. 2002.

Lei, Y. and P.J. Budden. "Limit Load Solutions for Thin-Walled Cylinders With Circumferential Cracks Under Combined Internal Pressure, Axial Tension and Bending." *Journal of Strain Analysis*. Vol. 39. pp. 673–683. 2004.

Microsoft Corporation. "Microsoft® SharePoint®." Redmond, Washington: Microsoft Corporation. 2010.

Miller, K.J. "The Short Crack Problem." *Fatigue of Engineering Materials and Structures*. Vol. 5. pp. 223–232. 1982.

NRC. Regulatory Guide 1.28, Rev. 4, "Quality Assurance Program Requirements (Design and Construction)." Washington, DC: NRC. 2010.

NRC. NUREG–1563, "Branch Technical Position on the Use of Expert Elicitation in the High-Level Radioactive Waste Program." Washington, DC: NRC. 1996.

NRC. NUREG–1297, "Peer Review for High-Level Nuclear Waste Repositories." Washington, DC: NRC. 1988.

O'Sullivan, B. "Mercurial: The Definitive Guide." Sebastopol, California: O'Reilly Media, Inc. June 2009.

Rahmann, S. "Net-Section-Collapse Analysis of Circumferentially Cracked Cylinders—Part II: Idealized Cracks and Closed-Form Solutions." *Engineering Fracture Mechanics*. Vol. 61. pp. 213–230. 1998.

Sandia National Laboratories. "xLPR Configuration Management Plan, R0." Albuquerque, New Mexico: Sandia National Laboratories. 2009.

Smith, R.A. "On the Short Crack Limitations of Fracture Mechanics." *International Journal of Fracture*. Vol. 13. pp. 717–720. 1977.

White, G., J. Broussard, J. Collin, M. Klug, C. Harrington, and G. DeBoo. "Advanced FEA Modeling of PWSCC Crack Growth in PWR Dissimilar Metal Piping Butt Welds and Application to the Industry Inspection and Mitigation Program." Proceedings of the PVP2008 ASME Pressure Vessels and Piping Division Conference, Chicago, Illinois, July 27–31, 2008. Paper No. PVP2008–61616. New York City, New York: American Society of Mechanical Engineers. 2008.

APPENDIX B

B1 VERIFICATION OF xLPR DETERMINISTIC MODULES

B1.1 Introduction

The purpose of this appendix is to present the verification results for the different deterministic modules contained within the extremely low probability of rupture (xLPR) code.

This section presents a brief overview of the xLPR code structure, verification approach, and testing platforms. Section B2 details the results of the verification testing performed on each xLPR module. Finally, Section B3 will present a summary of findings and conclusions with respect to the verification testing.

B1.1.1 The xLPR Code Structure

This section focuses on the modules that are called within the xLPR time loop as shown in Figure B–1 (Klasky, et al., 2010). As part of the xLPR probabilistic framework, the time loop is contained within the epistemic and aleatory loops [refer to U.S. Nuclear Regulatory Commission and Electric Power Research Institute (2011) for further details].

xLPR modules that were subjected to verification testing are listed in Table B–1. Note that the only module contained within the xLPR time loop that was not included in the verification testing was the leakage rate module, SQUIRT_v1.1.f90. This module was not included, because it is undergoing modifications by the xLPR working group.

The input and output for each module listed in Table B–1 will be detailed in Section B2.

B1.1.2 Verification Approach

The approach taken for performing the verification was to (i) spot check the source code by comparing the FORTRAN statements with the corresponding equations given in the appropriate references and (ii) spot check intermediate and final calculation results using hand calculations, spreadsheets, and third-party mathematics packages. In some cases, hand derivations were used to verify some of the expressions coded into a module. For example, this approach was used to verify the derivatives required for the Jacobian used in Newton's method in the module TWCFail_v2.1.f90.

Therefore, not all of the functionality was tested in each module. Numerical verification of a module was performed using either external FORTRAN programs as drivers or dynamic linked libraries (DLLs) driven by GoldSim® (GoldSim Technology Group LLC, 2010). For testing purposes, some FORTRAN modules were modified to generate an output file to print intermediate parameter values for verification of calculation results and execution sequence.

Verification of a module's numerical output was performed using hand calculations, Microsoft® Excel® (Microsoft Corporation, 2007) spreadsheets, or Mathematica™ (Wolfram Research Inc., 2008).

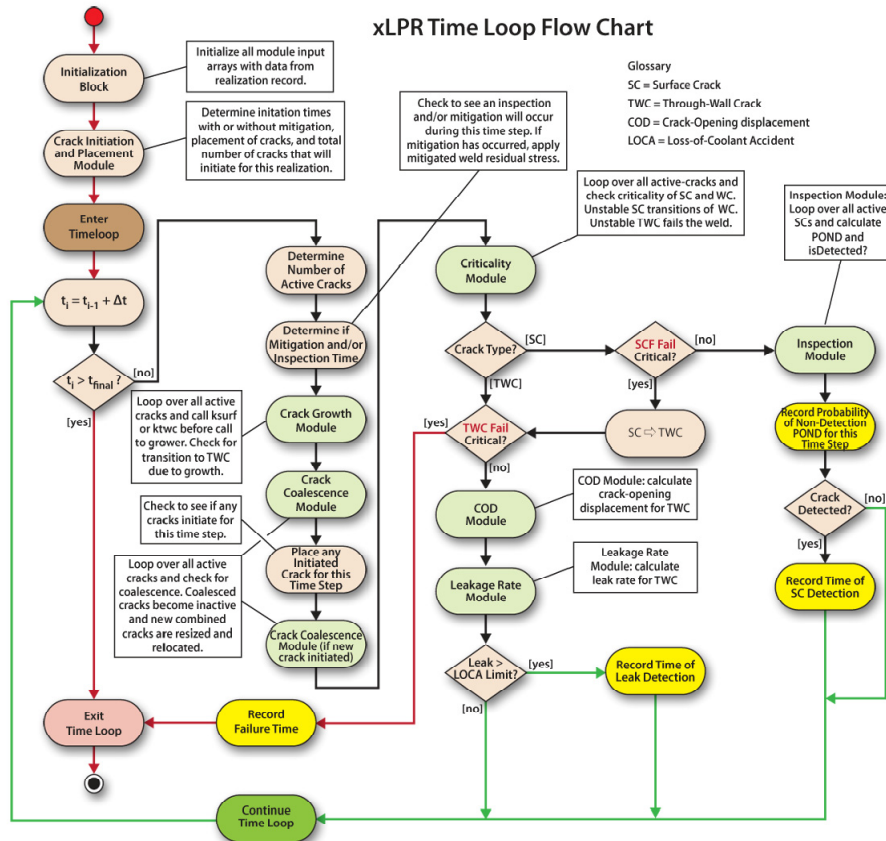


Figure B-1. xLPR Time Loop Flow Chart (Klasky, et al., 2010)

Table B-1. xLPR Modules	
Module Name	Module Function Description
crack_init_v2.1	Performs primary water stress corrosion cracking (PWSCC) crack initiation.
grower_v2.1	Calculates the PWSCC crack growth rate.
Coalescence_v2.2	Combines the cracks that meet the coalesce criterion based on Section XI, Article IWA-3000 of the ASME Boiler and Pressure Vessel code.*
kSurf_v1.1	Calculates the surface crack stress intensity, K, for a given crack size.
ktWC_v1.1	Calculates the K for a through-wall crack. Routine uses linear interpolation from tables of influence functions.
SCFail_v2.1	Calculates the maximum bending moment for a surface crack under pressure and axial load.
TWCFail_v2.1	Calculates the critical crack size (leading to pipe failure) given a bending moment.
COD_v2.1	Calculates the crack opening displacement for a given crack size in a pipe under prescribed axial and bending loads.
ISI_v2.1	Calculates the probability of not detecting a crack.
*ASME. "Boiler and Pressure Vessel Code." Article IWA-3000, Section XI. Standard Examination Evaluation. New York City, New York: The American Society of Mechanical Engineers. 2007.	

B1.1.3 Testing Platforms

The verification testing was performed on personal computers running either the Microsoft Windows® 7 or Windows XP operating system. The Lahey/Fujitsu FORTRAN 95 (Lahey Computer Systems, 2004) compiler was used for compiling source code, building executables (e.g., COD_V2.1 in Table B-1), and building DLLs.

B2 xLPR MODULE VERIFICATION TESTING

This section details the verification testing for each of the xLPR modules that are listed in Table B–1. In the sections that follow, individual tables list each module’s input and output variables, references used for FORTRAN code verification, and results of the verification testing.

B2.1 Verification of Crack Initiation, Growth, and Coalescence Modules

The software modules that were tested are listed in Table B–2. Input values and calculation results from each module were checked against reference documents. This checking was performed on a limited basis. The DLLs were modified to write input information and intermediate calculation results to output files. The results from the calculations and the input information were compared to the source documentation. The references used in this testing for each module are also shown in Table B–2.

B2.1.1 Crack Initiation

The crack initiation module, crack_init_v2.1.f90, contains three separate internal models—two labeled as direct models and one identified as a Weibull model. Each model was tested, and observations from this testing are listed in Table B–3. For each model, the equations in the code were compared to the equations in Harrington, et al. (2011) and the calculation results for the module were compared to spreadsheet-calculated values. Each observation is classified in terms of its effect on the documentation or the code.

Table B–2. xLPR Modules Included in Verification			
Module	Dynamic Link Library	GoldSim Model File	References
Crack Initiation, Version 2.1	crack_init_v2.1.dllx	crack_init_test_cases_v2.1.gsm	Harrington, et al., 2011*
Crack Growth, Version 2.1	grower_DLL_v2.1.dllx	grower_v2.1_GS_file.gsm	Harrington, et al., 2011* Ahluwalia, 2007†
Crack Coalescence, Version 2.2	Coalescence_DLLx_v2.2.dllx	GS_Framework_Coalescence_v2.2.gsm	Harrington, et al., 2011*
*Harrington, C., F. Ammirato, B. Brust, D. Dedhia, E. Focht, M. Kirk, C. Lange, R. Olsen, P. Scott, D.J. Shim, and G. White. “Models and Inputs Selected for Use in the xLPR Pilot Study.” Product Identifier 1022528. Palo Alto, California: Electric Power Research Institute. 2011. †Ahluwalia, K. “Materials Reliability Program: Mitigation of PWSCC in Nickel-Base Alloys by Optimizing Hydrogen in the Primary Water (MRP-213).” Product Identifier 1015288. Palo Alto, California: Electric Power Research Institute. 2007.			

Table B–3. Crack Initiation Module Test Results		
Model	Observations	Classification
Direct Model 1	None	None
Direct Model 2	There is a discrepancy between Eq. 3.32 and Figure 3.31 in Harrington, et al. (2011)* with regard to the QoverR term. The term is either + or – QoverR.	Possible code error Inconsistency in documentation
	The operator in the statement “if [(Stress/SigYS-z) eq. 0.0] then” should be .le.	Minor code error
	Some round-off error in the calculation of “D” propagates to initiation time. Harrington, et al. (2011, Eq. 3.30) shows 2/3, but the code uses a value of 0.66.	Minor code error
Weibull Model	None	None
Overall	Parameters in [Harrington, et al. (2011, Eq. 3.26, Direct Model 1; Eq. 3.32, Direct Model 2; and Eq. 3.34, (Weibull Model)] do not match parameters in Harrington, et al. (2011, Table 3.14), so it is not clear where the equations are accurately implemented in the code. For example, the terms “(Tempr + 273)” in the code need to be clarified for consistency with Harrington, et al. (2011). In addition, similar calculations in the crack growth module use 273.15 instead of 273. The code and documentation should be reviewed for consistency.	Minor document clarification Minor code inconsistency
	For Tempr, Harrington, et al. (2011, Table 3.14) specifies values greater than 30 °C but the code checks for values less than 30 °C. It should instead check for values less than or equal to 30 °C.	Minor code error
	For the same temperature and stress input data, the three models produce significantly different results. Therefore, the parameter values listed in Harrington, et al. (2011, Table 3.14) may need to be reviewed.	Possible model development review needed
*Harrington, C., F. Ammirato, B. Brust, D. Dedhia, E. Focht, M. Kirk, C. Lange, R. Olsen, P. Scott, D.J. Shim, and G. White. “Models and Inputs Selected for Use in the xLPR Pilot Study.” Product Identifier 1022528. Palo Alto, California: Electric Power Research Institute. 2011.		

None of the observations listed in Table B–3 are expected to significantly change the module calculation results. Therefore, minor code errors and minor document clarifications are listed in this table. However, because each model produces significantly different results for the same input stress and temperature values, the appropriateness of model parameters and validity of these different models should be reviewed as discussed in Harrington, et al. (2011).

B2.1.2 Crack Growth

For the crack growth module, grower_DLL_v2.1.f90, the equations in the code were compared to the equations in Harrington, et al. (2011) and intermediate and final calculation results were compared to spreadsheet-calculated values. The equations in the code were compared to Harrington, et al. (2011, Eqs. 3.20 through 3.24). Additionally, Ahluwalia (2007) was used to clarify the implementation of Harrington, et al. (2011, Eq. 3.21). From this comparison, one error is identified in the code. Ahluwalia (2007) shows a sample calculation pertaining to Harrington, et al. (2011, Eq. 3.21) that uses the base 10 logarithm (i.e., \log_{10}) versus the natural logarithm (i.e., \ln). The code, however, incorrectly implements Harrington, et al. (2011, Eq. 3.21) by using the natural logarithm in the calculation. This calculation is for the difference in electrochemical potential between the Ni/NiO transition and the electrochemical potential at the current concentration of hydrogen (i.e., $\Delta ECP_{Ni/NiO}$). This error changes the crack growth rate significantly and therefore is classified as a major code error.

B2.1.3 Crack Coalescence

For the crack coalescence module, Coalescence_DLLx_v2.2.f90, the software control logic was compared to the descriptions in Harrington, et al. (2011) for two test cases. One test case involved the coalescence of two surface cracks, and the other involved the coalescence of surface and through-wall cracks (TWC). In addition, the resulting (i.e., coalesced) crack location and length were compared to spreadsheet-calculated values. For both test cases, the logic for coalescing cracks agreed with descriptions in Harrington, et al. (2011) and the resulting values for crack location and length agreed with spreadsheet-calculated values and the descriptions in Harrington, et al. (2011).

B2.2 Verification of the Surface and Through-Wall-Crack Stress Intensity Factor, Surface Crack Maximum Bending Moment, and Through-Wall-Crack Critical Crack Size Modules

The xLPR modules that were validated are listed in Table B–4. The source code for each module was compared with reference documents to verify consistency. Line-by-line code inspection was performed for all the subroutines in the module. In most cases, mathematical derivations were performed by hand to verify that the corresponding FORTRAN statements agreed with the theory given in the appropriate references. Standalone programs for testing each module were obtained from the Microsoft SharePoint® configuration management (CM) directories. In all cases, the developer previously hardwired input values into the code. Calculations were verified by using FORTRAN write statements to generate intermediate calculations to a “debug” output file. This allowed validation of calculation sequence and calculation correctness. The output was also compared to documented results provided in an xLPR Model Document and Checking Desktop Guide provided for each module. These documents were obtained from the SharePoint CM directories.

B2.2.1 Stress Intensity Factor for a Surface Crack

This module is part of the overall Crack Growth module. In particular, kSurf_v1.1.f90 calculates the stress intensity factor for surface cracks. Within this module are calls to other subroutines, which calculate the influence functions for a number of (c/a) , (r/t) , and (a/t) ratios, where c is the crack half length, a is the crack depth, r is the pipe inner radius, and t is the wall thickness. On the basis of the influence functions, the surface crack tip stress intensity at the deepest point, surfk90, and the surface crack tip stress intensity at a surface point, surfk0, are calculated.

Table B–5 lists the inputs and outputs for kSurf_v1.1.f90.

Table B–4. xLPR Modules Included in Verification		
Module Calculation	Subroutine	References
Stress intensity factor, surface crack	kSurf_v1.1.f90	Excel® file “Surface Crack Anderson.xlsx” WRC Bulletin 471 (Anderson, et al., 2002)† ConceptualDescription-XLPR-DSK-004R0_SIAM_kSurf_v1.1.docx
Stress intensity factor, through-wall crack	KTWC_v1.1.f90	Excel file “TWC Anderson.xlsm” ConceptualDescription-XLPR-DSK-004R0_SIAM_kTWC_v1.1.docx
Bending moment for a surface crack	SCFail_v2.1.f90	Rahman (1998)‡ NUREG/CR • 4853 (Burst, 1987)§ ConceptualDescription-XLPR-DSK-004R0_SIAM_SCFail_v2.1.docx
Critical crack size (through-wall crack) leading to pipe failure	TWCFail_v2.1.f90	Rahman (1998) ‡ Gilles and Brust (1991) Brust and Gilles (1994)¶ NUREG/CR • 4853 (Brust, 1987)§ ConceptualDescription-XLPR-DSK-004R0_SIAM_TWCFail_v2.1.docx
<p>*The listed Excel files were obtained from the SharePoint® site.</p> <p>†Anderson, L., G. Thorwald, D.J. Reville, D.A. Osage, J.L. Janelle, and M.E. Fuhry. “Development of Stress Intensity Factor Solutions for Surface and Embedded Cracks in API 579.” <i>WRC Bulletin 471</i>. Shaker Heights, Ohio: Welding Research Council. 2002.</p> <p>‡Rahman, S. “Net-Section-Collapse Analysis of Circumferentially Cracked Cylinders—Part II: Idealized Cracks and Closed-Form Equations.” <i>Engineering Fracture Mechanics</i>. Vol. 61. pp. 213–230. 1998.</p> <p>§Brust, F.W. NUREG/CR–4853, “Approximate Methods for Fracture Analysis Through-Wall Cracked Pipes.” Washington, DC: NRC. pp. 2–13. 1987.</p> <p>¶Gilles P. and F.W. Brust. “Approximate Fracture Methods for Pipes—Part I: Theory.” <i>Nuclear Engineering and Design</i>. Vol. 127. pp. 1–17. 1991.</p> <p>¶Brust, F.W. and P. Gilles. “Approximate Methods for Fracture Analysis of Tubular Members Subjected to Combined Tensile and Bending Loads.” <i>Journal of Offshore Mechanics and Arctic Engineering</i>. Vol. 116. pp. 221–227. 1994.</p>		

Table B-5. Inputs and Outputs for Stress Intensity Module: kSurf_v1.1.f90		
Subroutine	Inputs	Outputs
kSurf_v1.1.f90	Pipe inner radius, Ri	Surfk90, surface crack tip stress intensity at deepest point
	Crack half-length, c	
	Crack depth, a	Surfk0, surface crack tip stress intensity at surface point
	Wall thickness, t	
	Membrane stress, sig0	
	Components of through-thickness stress, sig1–sig4	
	Bending stress, sig5	

The subroutine, calcK (called by kSurf), determines the surface crack stress intensity factor, K, for a given crack size. Specifically, surfk90 is the crack tip stress intensity at the deepest point and surfk0 is the crack tip stress intensity at a surface point.

The calcK subroutine calls a number of other subroutines to calculate the appropriate influence functions based upon the (c/a), (r/t), and (a/t) ratios. For an (a/c) ratio greater than 0.2 calcK calls the subroutine calcG. For an (a/t) ratio less than 0.2, calcK calls the subroutines calcG and calcGaovert0. Specifically, where (a/t) is less than 0.2, calcK calls subroutine calcG for an (a/t) equal to 0.2 and calls the subroutine calcGaovert0 for an (a/t) equal to 0. Subsequently, linear interpolation (subroutine lininterp) is performed using the solutions at (a/t) equal to 0 and (a/t) equal to 0.2 to determine the influence functions for the actual (a/t) ratio between 0 and 0.2.

For a specified (a/t) ratio, subroutine calcG calculates the influence functions for different (c/a) and (r/t) ratios. The FORTRAN code for calcG is straightforward and consists of a series of FORTRAN expressions used to calculate each influence function depending on the specific (c/a) and (r/t) ratio. Each of these expressions was verified line-by-line by comparing with the influence function coefficients given in the Excel file Surface Crack Anderson.xlsx (obtained from the SharePoint CM site) with those from the Influence Function Equations and Influence Coefficients given in Anderson, et al. (2002). The coefficients in the code matched those in the references.

The subroutine calcG determines influence functions for values of (c/a) between 1 and 32. If the crack's (c/a) ratio is larger than 32, then calcG calls the subroutine calcGinfinity, which calculates the influence function for a given (r/t) and (a/t) ratio. All of the coding in this module was compared with the corresponding expressions given in Anderson, et al. (2002), and the code was verified to match the expressions.

A standalone version of kSurf_v1.1.f90 was obtained from the SharePoint CM site and used for the numerical tests. The test procedure utilized FORTRAN write statements placed in subroutines calcG, calcGaovert0, and calcGinfinity. All of the output was written to a debug file. The output to this file allowed the calculation sequence to be verified for different (c/a), (r/t), and (a/t) ratios; the correct passing of arguments; and specific values of the influence functions. The subroutine lininterp is the same as that called in the module kTWC_v1.1.f90 and will be discussed in that section.

The numerical output values of selected FORTRAN expressions were evaluated using simple hand calculations by substituting the numerical value of each variable into the FORTRAN statement and verifying the result that was printed in the debug file. All numerical hand calculations agreed with those printed in the debug file.

For the calculations that were spot checked, all of the calculations agreed with the expected results as documented in the check file obtained from the SharePoint CM site:

ConceptualDescription-XLPR-DSK-004R0_SIAM_kSurf_v1.1.docx (Williams and Klasky, 2010a). This verification against the check file was done to confirm the code that was verified, matched the documented check file results.

B2.2.2 Stress Intensity Factor for a Through-Wall Crack

This subroutine is part of the overall Crack Growth module. The module kTWC_v1.1.f90 calculates the stress intensity factor for a TWC by using linear interpolation from tables of influence functions.

Table B–6 lists the inputs and outputs for kTWC_v1.1.f90.

The main routine kTWC calls subroutine calcTWCK, which is used for calculating the stress intensity factor, K , for a TWC using linear interpolation from tables of influence functions. Subroutine calcTWCK calls the subroutine calcGTWC to calculate the necessary coefficients used in the influence functions for a TWC. In specific cases, this subroutine utilizes the subroutine lininterp, which performs simple linear interpolation.

Table B–6. Inputs and Outputs for Stress Intensity Module: kTWC_v1.1.f90		
Subroutine	Inputs	Outputs
kTWC_v1.1.f90	Pipe inner radius, R_i	TWCK, through-wall crack tip stress intensity
	Crack half-length, c	
	Pipe thickness, $thick$	
	Total axial stress, $sig0$	
	ID WRS from load module, $sig0_wrs$	
	Local bending stress at the crack, $sig5$	

The purpose of these routines is to populate an array with the influence functions. The array contains data for specific ($R_i/thick$) ratios. For a ($R_i/thick$) ratio other than that directly having available data, simple linear interpolation is used to calculate influence function values for the given ($R_i/thick$) ratio.

Subroutine calcGTWC basically uses data tables of influence functions to populate the necessary arrays, which are used to calculate the influence function coefficients $g0twc$, $g1twc$, and $g5twc$ and are the output from calcGTWC. The routine calcTWCK uses the output of calcGTWC to calculate the TWC tip stress intensity, TWCK.

Because the basic function of the routines is to perform a “look up” of data from tables, the source code of all of the routines was checked by using line-by-line inspection. The verification of the coding for the linear interpolation algorithm was compared to numerical algorithms [e.g., those given in Press, et al. (1992).] The algorithm in the software is similar to other algorithms involving a look up of data.

The influence function data were verified by comparing the data statements in calcGTWC with the data given in the Excel file “TWC Anderson.xlsm,” which was obtained from the SharePoint CM site. All of the data were verified to be consistent with “TWC Anderson.xlsm.”

A standalone version of kTWC_v1.1.f90 was obtained from the SharePoint CM site and used for the numerical tests. The test procedure utilized FORTRAN write statements placed primarily in calcGTWC and lininterp. These write statements were used to output the values of specific variables. All of the output was written to a debug file, which allowed the calculation sequence to be verified, as well as the passing of arguments. The FORTRAN write statements were placed at specific locations to print out components of the array $a(.....)$ to verify that the

correct data were being copied from the declared FORTRAN data statement. The passing of data and the calculation sequence were both verified to be correct. In addition, a value of (Ri/thick), variable name rovert in the code, was chosen such that the routine lininterp was used. The routine lininterp is a straightforward implementation of linear interpolation and was verified to be calculating the interpolated value correctly.

For the calculations that were spot checked, all of the calculations printed in the debug file agreed with the expected results for the stress intensity factor documented in the check file obtained from the SharePoint CM site: ConceptualDescription-XLPR-DSK-004R0_SIAM_kTWC_v1.1.docx (Williams and Klasky, 2010b). This verification against the check file was done to confirm that the code matched the documented checkfile results.

B2.2.3 Maximum Bending Moment for a Surface Crack

The module SCFail_v2.1.f90 calculates the maximum bending moment for a surface crack subject to pressure and axial loads.

Table B–7 lists the inputs and outputs for SCFail_v2.1.f90.

The main routine SC_Fail calls two other subroutines. The subroutine BM_CD_NSC determines the maximum moment for a constant depth surface crack based on Net Section Collapse (NSC). This subroutine considers two cases: (i) the entire crack in the tension zone and (ii) part of the crack in the compression zone. For case (ii), there are checks to determine whether or not there is crack closure. As given in Rahman (1998), because the crack has a constant depth, closed-form solutions are available to calculate the stress-inversion angle, β , and the NSC moment. There are no calls to other functions within this subroutine.

The subroutine BM_SESC_NSC determines the maximum moment for a semi-elliptical surface crack based on NSC. This subroutine utilizes the solutions given in Rahman (1998). This subroutine also considers two cases: (i) the entire crack in the tension zone and (ii) part of the crack in the compression zone. For case (ii), there are checks to determine whether or not there is crack closure.

For case (i) where the entire crack is in the tension zone, Rahman (1998) provides closed-form solutions for the stress-inversion angle, β , and the NSC moment. In support of these calculations, calls are made to the functions FN_GAMMA and FN_SUM1_SESC_NSC. The implementation of the FN_GAMMA function was checked with NUREG/CR–4853 (Burst, 1987) and verified to be correct. The implementation of function FN_SUM1_SESC_NSC was checked with Rahman (1998) and verified to be correct.

However, for case (ii), closed-form solutions do not exist. As given in Rahman (1998), the value for β must be determined numerically. The function FN_FIND_BETA uses Newton's Method to find a solution for β . The majority of the terms in this function were checked directly with Rahman (1998) and verified to be correct. A single derivative was verified by simple hand derivation. The subroutine BM_SESC_NSC also uses the function FN_SUM2_SESC_NSC. During code verification it was noted that the source code stated the code had been changed to reflect an error given in Rahman (1998). This error correction was also documented in Harrington, et al. (2011).

Table B-7. Inputs and Outputs for Net Section Collapse Module: SCFail_v2.1.f90		
Subroutine	Inputs	Outputs
SCFail_v2.1.f90	Pipe outer radius, R_o	BM_Ratio, bending moment ratio
	Crack depth, depth	
	Pipe wall thickness, thick	
	Half-crack length, theta	
	Pipe yield stress, sigy	
	Pipe ultimate stress, sigu	
	Applied bending moment, BM	
	Pipe internal pressure, pressure	
	Applied axial force, F_ax	
	SC analysis method: =0, constant depth surface crack =1, semi-elliptical surface crack	

A standalone version of SCFail_v2.1.f90 was obtained from the SharePoint CM site and used for the numerical tests. The test procedure utilized FORTRAN write statements placed throughout all of the subroutines and functions used in SCFail_v2.1. These write statements were used to output the values of specific variables. All of the output was written to a debug file, allowing the calculation sequence and the passing of arguments to be verified. The passing of data and the calculation sequence were both verified to be correct. The values of selected FORTRAN expressions were evaluated using hand calculations by substituting the numerical value of each variable and verifying the result. For the functions FN_SUM1_SESC_NSC, FN_SUM2_SESC_NSC, and FN_FIND_BETA, by examining the debug file, the numerical iterations were verified to converge within the specified tolerance.

For the calculations that were spot checked, all of the calculations printed in the debug file agreed with the expected results for the maximum bending moment documented in the check file obtained from the SharePoint CM site: ConceptualDescription-XLPR-DSK-004R0_SIAM_SCFail_v2.1.docx (Williams, 2010a). This verification against the checkfile was done to confirm that the code matched the documented checkfile results.

B2.2.4 Critical Crack Size for a Through-Wall Crack

The module TWCFail_v2.1.f90 calculates the critical crack size leading to pipe failure for a specified bending moment.

Table B-8 lists the inputs and outputs for TWCFail_v2.1.f90.

The main routine TWC_fail determines whether a TWC will fail under an applied load. The TWC_fail routine calls two subroutines: THETA_CD_NSC and ENG2_mp.

Table B–8. Inputs and Outputs for Critical Crack Size Module: TWCFail_v2.1.f90		
Subroutine	Inputs	Outputs
TWCFail_v2.1.f90	Pipe outer radius, R_o	Theta_r, a measure of closeness to failure in terms of critical crack size If_flag, failure and failure mode indication flag
	Pipe wall thickness, thick	
	Half crack length, theta	
	Pipe yield stress, sigy	
	Pipe ultimate stress, sigu	
	Ramberg-Osgood coefficient, RO_alpha	
	Ramberg-Osgood reference stress, RO_sigo	
	Ramberg-Osgood reference strain, RO_epso	
	Ramberg-Osgood exponent, RO_n	
	Pipe material initiation J-resistance, Resist_Jic	
	Pipe material J-resistance coefficient, Resist_C	
	Pipe material initiation J-resistance exponent, Resist_m	
	Applied bending moment, BM	
	Pipe internal pressure, pressure	
	Applied axial force, F_ax	

The subroutine THETA_CD_NSC determines the critical crack size for a constant depth TWC using NSC assumptions. The equations used in this subroutine are based upon the theory given in Rahman (1998) in which closed-form solutions are given for NSC. Rahman (1998, Eqs. 12 and 13) was used to determine the critical crack length, 2θ . Based upon the form of Rahman (1998, Eqs. 12 and 13), an iterative solution using Newton's Method is necessary to solve for θ . All of the FORTRAN expressions used in subroutine THETA_CD_NSC were checked directly with Rahman (1998) and verified to be correct. A single derivative necessary for Newton's Method was verified by hand derivation. There are no calls to other subroutines or functions within THETA_CD_NSC.

The subroutine ENG2_mp determines the equilibrium crack angle for a TWC for a pipe subjected to internal pressure and bending moment. This calculation uses the LBB.ENG2 method (Brust and Gilles, 1994), which is based on elastic plastic fracture mechanics (EPFM). ENG2_mp calls a number of other subroutines and functions in support of the EPFM calculations. The calculations performed in the subroutine ENG2_mp are based upon a J-estimation scheme technique. This method is detailed in Gilles and Brust (1991), Brust and Gilles (1994), and NUREG/CR-4853 (Brust, 1987). All of the FORTRAN expressions used in subroutine ENG2_mp were checked directly with these references and were verified to be correct. However, note in the FORTRAN source code there are comments pointing out errors in the referenced equations that have been corrected in the source code for the subroutines ENG2_mp and CalcJ_Stuff, and in the functions FN_Itlb2, FN_dltlb2_dtheta, FN_dltlb2_dtheta2, FN_Itlb, FN_dltlb_dtheta, FN_d2ltlb_dtheta2, FN_dFtFb_dtheta, and FN_d2FtFb_dtheta2. These corrections were verified based upon documentation given in Harrington, et al. (2011).

A standalone version of TWCFail_v2.1.f90 was obtained from the SharePoint CM site and used for the numerical tests. The test procedure utilized FORTRAN write statements placed throughout all of the subroutines and functions used in TWCFail_v2.1. These write statements were used to output the values of specific variables. All of the output was written to a debug file, which allowed the calculation sequence and the passing of arguments to be verified. The passing of data and the calculation sequence were also verified to be correct. The values of

selected FORTRAN expressions were evaluated by simple hand calculations by substituting the numerical value of each variable and verifying the result.

- For the subroutine THETA_CD_NSC, for the calculations that were spot checked, all of the calculations agreed with the expected results. The Newton Method iteration loop was checked by printing intermediate values. The numerical iterations were verified to converge within the specified tolerance and terminate correctly.
- Intermediate calculations were spot checked for the subroutine ENG2_mp, and all output values agreed with the expected results. One Newton iteration loop was checked by verifying convergence.
- ENG2_mp calls the subroutines CALC_STRESS and CalcJ_Stuff. The numerical calculations for both subroutines were verified by examining the debug output file.
- For the functions FN_SUM1_SESC_NSC, FN_SUM2_SESC_NSC, and FN_FIND_BETA, by examining the debug file, the iterations were verified to converge within the specified tolerance by examining the debug file.

For the calculations that were spot checked, all of the calculations printed in the debug file agreed with the expected results for the critical TWC size documented in the check file obtained from the SharePoint CM site: ConceptualDescription-XLPR-DSK-004R0_SIAM_TWCFail_v2.1.docx (Williams, 2010b). This verification against the checkfile was done to confirm that the code matched the documented checkfile results.

B2.3 Verification of the Crack Opening Displacement Module

The software module that was part of this review is the crack opening displacement (COD), COD_v2.1.f90, module. Input values and calculation results from the module were checked against reference documents, as shown in Table B–9. A FORTRAN driver program (checkCODdriver.f90 and checkCODdriver.exe) was written to call the appropriate subroutines and carry out the various tests.

B2.3.1 Crack Opening Displacement

The use of the GE/Electric Power Research Institute (EPRI) method to calculate COD is requested through a parameter, “method,” when calling the Calc_COD subroutine. At present, the GE/EPRI method for calculating COD is available in Mattie, et al. (2010).

The expressions used in the COD_v2.1.f90 module appear consistent with those detailed in the “xLPR Model Document and Checking Desktop Guide” (Olson, 2010). As described there, the calculation implemented for the GE/EPRI method used the blended solution coded in NRCPIPE Version 3.0 (Battelle, 1996).

Table B–10 lists the inputs and outputs for the COD_v2.1.f90 module.

Note Olson (2010) and Mattie (2010) state that there is a mistake in the implementation of the solution in NRCPIPE (Battelle, 1996) for the blended GE/EPRI COD solution. Olson (2010) indicates that COD_v2.1.f90 uses a slightly modified form of the equations that does not include the referenced mistake. The COD_v2.1.f90 module uses this modified form of the equations.

As part of the spot-checking approach, several subroutines, functions, and individual expressions from the COD_v2.1.f90 module were tested to ensure that appropriate output was returned. For example, using the checkCODdriver.exe program, the following subroutines were checked: (i) Calc_COD, (ii) Blended_COD, and (iii) GetNeighborIndices. Verification of these subroutines involved running a number of test cases, each with different input values.

Table B–9. xLPR Module Included in Verification		
Module Calculation	Subroutine	References
Crack opening displacement	COD_v2.1.f90	Olson (2010);* Mattie (2010)†
*Olson, R. "xLPR Model Document and Checking Desktop Guide, Model ID: COD_v2.0." Columbus, Ohio: Battelle. 2010.		
†Mattie, P.D. "xLPR Model Document and Checking Desktop Guide, Model ID: COD_v2.1.DLL." Albuquerque, New Mexico: Sandia National Laboratories. 2010.		

Table B–10. Inputs and Outputs for Crack Opening Displacement Module: COD_v2.1.f90			
Inputs		Outputs	
Diameter =	R_o: pipe outer radius	Crack opening displacement =	COD_OD: crack opening on the outside of the pipe at the centerline of the crack (-1.0 if the if_flag < 0)
Wall thickness =	thick: pipe wall thickness		
Crack length =	theta: half-crack length (radians)		COD_ID: crack opening on the inside of the pipe at the centerline of the crack (-1.0 if the if_flag < 0)
Operating Loads =	pressure: pipe internal pressure		
Transient Loads =	BM: applied bending moment		
	F_ax: applied axial force (excluding pressure effect)		
Material Flow Stress (Stress Strain) =	sigy: pipe yield stress	Other =	if_flag: failure indication flag
	sigu: pipe ultimate stress		
	RO_alpha, RO_sigo, RO_epso, RO_n: Ramberg-Osgood stress-strain parameters		
	sig0_wrs, sig1_wrs, sig2_wrs, sig3_wrs: Residual stress polynomial terms 0 through 3		
Other =	Resist_Jic, Resist_C, Resist_m: J-R curve constants (failure criteria)		
	method: COD analysis method selector		

Several input values for parameters were imported from a text file [testinputdata.txt] using the FORTRAN driver program. Several parameter calculations contained in some of the subroutines were also verified by insertion of intermediate dumps of data to the test logging file [parametervalueDump.log]. Additionally, a Mathematica™ notebook was used to compute independent parameter values for several test cases [see check_COD_v2.1.nb]. The test driver, calculation, and logging files (listed in Table B–11) are included together in the archive file [codchecks.zip].

The subroutine Calc_COD is called using parameter values from the six test cases described in the xLPR Model Document and Checking Desktop Guide (Mattie, 2010). These cases were selected because the values can be readily compared to those previously tested and the input parameters cover a range of representative data. The parameter input values for the test cases are imported into the FORTRAN driver program (checkCODdriver.exe) from a text file (testinputdata.txt). All test cases are called using method 0 for the GE/EPRI calculation. The input values used for test cases 1 to 5 are given in Table B–12.

Table B-11. Files Contained in the Archive <i>codchecks.zip</i>	
File Name	Notes
checkCODdriver.f90	This is the test driver program, which calls several subroutines from the COD_v2.1.90 module.
checkCODdriver.exe	The check driver executable program. The executable file was built linking both the checkCODdriver.f90 and COD_v2.1_mod4checks.f90 object code.
testinputdata.txt	This contains the input values for the test cases. When checkCODdriver.exe is executed, this input file should be used.
COD_v2.1_mod4checks.f90	This is the original COD_v2.1.90 source code, which has had additional debugging statements added to access intermediate parameter values.
parametervalueDump.log	This log file is used to store and view parameter values from the additional debugging statements that were added to the original COD_v2.1.f90 source code.
check_COD_v2.1.nb	This is a Mathematica™ notebook that contains several additional calculations and checks for the subroutines and parameters.
check_COD_v2.1.pdf	This is a pdf format file containing the contents of a Mathematica™ notebook file (for user convenience).

Table B-12. Parameter Input Values for Test Cases 1 to 5 as Presented in Olson (2010)*					
Parameter, Units	Test Case 1	Test Case 2	Test Case 3	Test Case 4	Test Case 5
R_o, in	2.25	18	2.25	6.375	2.1
thick, in	0.214285714	1.714285714	0.214285714	0.607142857	0.2
theta, rad	0.5013974	1.0027955	1.002795267	1.002795459	1.0547
pressure, psi	933.3333333	933.3333333	807.1428571	807.1428571	0
BM, in-lb	130597	22176061.93	44860.27408	930047.6436	325242.222
F_ax, lb	0	0	0	0	0
sigy, psi	49983.25567	49983.25567	35789.61475	35789.61475	50000
sigu, psi	74974.8835	74974.8835	47013.58238	47013.58238	110000
RO_alpha, -	6.349384579	6.349384579	11	11	1
RO_sigo, psi	49983.25567	49983.25567	35789.61475	35789.61475	50000
RO_epso, -	0.001785116	0.001785116	0.001350552	0.001350552	0.001666667
RO_n, -	6.316480712	6.316480712	4.754679198	4.754679198	2
Resist_Jic, lb/in	0	0	0	0	14515.93603
Resist_C, -	0	0	0	0	105374.9917
Resist_m, -	0	0	0	0	1.03491313
sig0_wrs, psi	0	0	0	0	0
sig1_wrs, psi	0	0	0	0	0
sig2_wrs, psi	0	0	0	0	0
sig3_wrs, psi	0	0	0	0	0

*Olson, R. "xLPR Model Document and Checking Desktop Guide, Model ID: COD_v2.0." Columbus, Ohio: Batelle. 2010.

The parameter input values for test case 6 are given in Table B-13. Note that for Test Case 6, Olson (2010) gives the input vales in SI units.

Table B–13. Parameter Input Values for Test Case 6 as Presented in Olson (2010)*	
Parameter, Units	Test Case 6
R_o, mm	190.5
thick, mm	40.132
theta, rad	1.8
pressure, MPa	15.52989
BM, N-mm	206879000
F_ax, N	21570
sigy, MPa	124.322
sigu, MPa	368.641
alpha, -	4.47291
sigo, MPa	124.322
epso, -	0.000701989
n, -	3.93325
J_ic, N/mm	391.884
C	225.528
m, -	0.622615
sig0_wrs, MPa	286.06
sig1_wrs, MPa	-1080.1
sig2_wrs, MPa	-1.06342
sig3_wrs, MPa	1017.37
COD, mm	334.151111

*Olson, R. "xLPR Model Document and Checking Desktop Guide, Model ID: COD_v2.0." Columbus, Ohio: Batelle. 2010.

Table B–14 provides the expected results for the test case calculations as reported in Olson (2010).

Note that the listed results from Mattie (2010) are for the COD and are collected from several source programs. The results listed as "Program" are from the COD_v2.1 module as tested by Olson (2010), the results from Excel are from an independent spreadsheet (Mattie, 2010), and the results listed as "NRCPIPE" are from the NRCPIPE User's Guide (Battelle, 1996).

Table B–14. Expected Results from Olson (2010)*						
Parameter, Units	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6
Program	0.0655901 in	0.1938032 in	0.0861614 in	0.176766 in	0.5620454 in	334.151111 mm
Excel®	0.065591 in	0.1938032 in	0.0861613 in	0.176766 in	0.5620452 in	—
NRCPIPE	na	na	na	na	0.5618 in	—

*Olson, R. "xLPR Model Document and Checking Desktop Guide, Model ID: COD_v2.0." Columbus, Ohio: Batelle. 2010.

Table B–15 shows the result values that were calculated for the six test cases.

The results for all of the test cases match those previously reported from the COD_v2.1 module (Mattie, 2010) by rounding the final digits. These results agree well with the results reported from Excel (Mattie, 2010) and with the values reported from NRCPIPE (Battelle, 1996).

For the subroutine Blended_COD, a number of internal parameters were checked. The variable h2_tdata is equated to the results from a call to the GetBinlinearInterpolation function. The inputs to this function are the parameters TOP_h2t10, en_h2t10, h2_t10data, aob, and RO_n. The values for TOP_h2t10, en_h2t10, and h2_t10data are defined in the COD module as arrays, and their values are constant. The values for aob and RO_n are varied for each case tested. RO_n was an input parameter for the Calc_COD subroutine, and aob is calculated internally. The values used for the six input test cases were the same as those used for Test 1 of subroutine Calc_COD.

To determine the expected results for the GetBilinearInterpolation function (h2_tdata), a separate Mathematica™ notebook was written to calculate the appropriate interpolated values (see check_COD_v2.1.nb for more information). Values of aob, calculated in the Calc_COD module for the test cases 1 to 6, were extracted from test case runs by exporting values to an external text file. Table B–16 gives the input values for aob and RO_n used in the test cases.

The expected results shown in Table B–17 are from calculations using Mathematica™ and documented in the notebook (check_COD_v2.1.nb).

Table B–18 gives the results obtained from the checkCODdriver program for the six test cases.

As shown, the test results for all cases agree with the expected results considering possible round-off error.

The crack opening displacement due to the tensile axial force is defined by the parameter delta_pt and is calculated in subroutine Blended_COD. The expression used to calculate delta_pt is detailed in Mattie (2010). The expression for delta_pt coded in the subroutine Blended_COD is the second term in the expression defined in Mattie (2010, Eq. 3). There, the author notes that there was a mistake in implementing the solution in NRCPIPE (Battelle, 1996). Mattie (2010) refers to GE/EPRI NP-5596 (EPRI, 1988) and indicates that the π in the P_{tot}/P_0 term does not belong there. This adjustment to the expression is further noted in the COD_v2.1.f90 source code.

Table B–15. Calculated Results for All 6 Test Cases						
Parameter, Units	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6
check CODdriver.exe Program results	0.06559006 in	0.19380319 in	0.08616136 in	0.1767660 in	0.56204536 in	334.1511111 mm

Table B–16. Input Values for aob and RO_n		
Case #	Parameter	
	aob	RO_n
1	0.159599749326848	6.316480711999999
2	0.319199721470617	6.316480711999999
3	0.319199647304414	4.754679197999999
4	0.319199708419912	4.754679197999999
5	0.335721436958044	2.000000000000000
6	0.572957795130823	3.93325

Table B–17. Output Values for aob and RO_n Using Mathematica™			
Case	aob	RO_n	Interpolated Results
1	0.159599749326848	6.316480711999999	4.640597114534663
2	0.319199721470617	6.316480711999999	2.5303812910481627
3	0.319199647304414	4.754679197999999	3.103644802818521
4	0.319199708419912	4.754679197999999	3.1036443056682113
5	0.335721436958044	2.	4.387134900018311
6	0.572957795130823	3.93325	2.04444175

Table B–18. Output Values for aob and RO_n			
Case #	Parameter		Test Result h2_tdata(2,1)
	aob	RO_n	
1	0.159599749326848	6.316480711999999	4.640597115
2	0.319199721470617	6.316480711999999	2.530381291
3	0.319199647304414	4.754679197999999	3.103644803
4	0.319199708419912	4.754679197999999	3.103644306
5	0.335721436958044	2.000000000000000	4.3871349
6	0.572957795130823	3.933250000000000	2.04444175

For the verification testing, the values for the parameter `delta_pt` were calculated independently in a separate Mathematica™ notebook (see `check_COD_v2.1.nb`), based upon the expressions presented in the xLPR Model Documentation and Checking Desktop Guide (Mattie, 2010) and the `COD_v2.1.f90` source code. The values for `delta_pt` were calculated for the six test cases considered in the test of subroutine `Calc_COD`. The test cases and expected results (based upon parameter value dumps from the `COD_v2.1` module) are given in Table B–19.

The results from the Mathematica™ notebook, as shown in Table B–20, agree with those from intermediate values extracted from the `COD_v2.1` module. Therefore, all tests pass to within expected precision.

Table B–19. Expected Results for the Parameter <code>delta_pt</code> from <code>COD_v2.1</code> Module	
Case #	<code>delta_pt</code>
1	1.000448727E-07
2	2.47842975E-05
3	1.412412089E-04
4	4.00183894E-04
5	0
6	24.630017

Table B–20. Comparison of Results for Parameter <code>delta_pt</code>		
Case #	<code>COD_V2.1</code> Module Output	Mathematica Output
1	1.000448727E-07	1.0004487271486431E-07
2	2.47842975E-05	2.4784297594405805E-05
3	1.412412089E-04	1.4124120894351E-04
4	4.00183894E-04	4.001838940377014E-04
5	0	0
6	24.630017	24.63001707844345

The parameter `delta_pbpt` is calculated in subroutine `Blended_COD`. For the test, the values for the parameter `delta_pbpt` were calculated independently in a separate Mathematica™ notebook (see `check_COD_v2.1.nb`), based upon the expressions presented in the xLPR Model Documentation and Checking Desktop Guide (Mattie, 2010) and the `COD_v2.1.f90` source code. The values for `delta_pbpt` were calculated for the six test cases considered in the test for Subroutine `Calc_COD`. The test cases and expected results (based upon parameter value dumps from the `COD_v2.1` module) are given in Table B–21.

The results from the Mathematica™ notebook, given in Table B–22, agree with those from intermediate values extracted from the `COD_v2.1` module. Therefore, all tests pass to within expected precision.

The function `GetBilinearInterpolation` is called from subroutine `Blended_COD`. In this test, the subroutine `GetBilinearInterpolation` is called five times. The five tests are intended to verify that for a given, simple set of input coordinates, the function will properly handle requests for indices that lie outside of the range of values in the indices, i.e., exactly on the indices, having one value on the indices with another value between indices, and having both values between indices. The test indices (coordinates) used are given in Table B–23.

Table B-21. Expected Results for Parameter delta_pbpt from COD_v2.1 Module	
Case #	delta_pbpt
1	5.371345E-02
2	1.642915E-02
3	6.1857778977E-02
4	0.11280231759
5	0.430521175
6	—

Table B-22. Comparison of Results for Parameter delta_pbpt		
Case #	COD_v2.1 Module Output	Mathematica Output
1	5.371345E-02	5.371345078046241E-02
2	1.642915E-02	1.6429150589244587E-02
3	6.1857778977E-02	1.85777897729749E-02
4	0.11280231759	0.11280231759279842
5	0.430521175	0.4305211754024092
6	—	—

Table B-23. Test Indices	
Parameter (Array)	Indice Coordinate Values
testxlist	1, 2, 3
testylist	1, 2, 3, 4, 5

The third dimension (z) is a function of the testxlist and testylist values. For this test, the z surface was defined as $z(x,y) = x \times y$; therefore, the values given in Table B-24 were used for the z surface.

The test case input values for the x and y coordinates, the assumed test indices, and the expected results from the function GetBilinearInterpolation are given in Table B-25 (as calculated in the Mathematica™ notebook check_COD_v2.1.nb).

The results for the five tests, as output from the checkCODdriver program (e.g., see parametervalueDump.log, TESTR1) were in agreement with all expected results as shown in Table B-26 (all tests passed).

The subroutine GetNeighborIndices is called from within Blended_COD. In this test, the subroutine GetNeighborIndices is called four times. The four tests are intended to verify that for a given one-dimensional array of values, the subroutine will properly handle requests for neighbor indices for values that lie between two indices, for values that lie outside of the range of the indices (both for values greater and less than the given range), and for a requested number that is the same value as one of the indices. The input values and expected results are given in Table B-27.

The results from the test, as obtained from the checkCODdriver.exe console output, are given in Table B-28. The expected results were observed for all trials; therefore, all tests pass.

B2.4 Verification of the In-Service Inspection Module

This xLPR module, ISI_v2.1.f90, calculates the probability of not detecting a crack for various crack depth to wall thickness ratios and returns a flag (0 = not detected, 1 = detected) that indicates whether or not the crack was detected. The software module that was part of this review is shown in Table B-29. Input values and calculation results from the module were checked against a reference document. This checking was performed on a limited basis. The DLLs were modified to write input information and intermediate calculation results to output files.

The results from the calculations and the input information were compared to the source documentation. The reference used in this testing for this module is also shown in Table B–29.

Table B–24. Values Used for Z Surface						
Array Elements						
Parameter (Array, Dimension (3, 5))	X Coordinate	Y Coordinates				
testzsurface		1	2	3	4	5
	1	1	2	3	4	5
	2	2	4	6	8	10
	3	3	6	9	12	15

Table B–25. Expected Results From GetBilinearInterpolation			
Test Case	x Coordinate Value (x#)	y Coordinate value (y#)	Expected Result
R1	0	0	1
R2	4	6	15
R3	2	3	6
R4	2.5	3	7.5
R5	2.5	3.5	8.75

Table B–26. Test Results From GetBilinearInterpolation			
Test Case	x Coordinate Value (x#)	y Coordinate Value (y#)	Test Result
R1	0	0	1
R2	4	6	15
R3	2	3	6
R4	2.5	3	7.5
R5	2.5	3.5	8.75

Table B–27. Expected Results of GetNeighborIndices Testing			
Test	Input Coordinate Value	Expected Lower Value Returned	Expected Greater Value Returned
3A	4.5	4	5
3B	11.0	10	10
3C	7.0	7	7
3D	0	1	1

Table B–28. Test Results of GetNeighborIndices			
Test	Input Coordinate Value	Lower Value Returned From Test	Greater Value Returned From Test
3A	4.5	4	5
3B	11.0	10	10
3C	7.0	7	7
3D	0	1	1

Table B–29. xLPR Module Included in Verification			
Module	Dynamic Link Library	GoldSim Model File	References
In-Service Inspection, Version 2.1	ISI_DLL_v2.1.dllx	ISI_v2.1_GS_Framework_Test_File.gsm	Harrington, et al. (2011)*
*Harrington, C., F. Ammirato, B. Brust, D. Dedhia, E. Focht, M. Kirk, C. Lange, R. Olsen, P. Scott, D.J. Shim, and G. White. "Models and Inputs Selected for Use in the xLPR Pilot Study." Product Identifier 1022528. Palo Alto, California: Electric Power Research Institute. 2011.			

Table B–30 shows the input and output for the in-service inspection module.

The software logic agreed with the descriptions in Harrington, et al. (2011), and results from the software agreed with spreadsheet calculations. However, the following is recommended:

1. Harrington, et al. (2011) indicate that if a random sample is less than the calculated probability of detection, then the crack is considered to have been detected. Revise Harrington, et al. (2011) to indicate that if the calculated probability of nondetection is

less than or equal to a randomly sampled value from a uniform distribution, then the crack is considered to have been detected.

2. Add error detection to the DLL. For example, if a crack depth to wall thickness ratio greater than one is input, then the DLL can still produce a probability of nondetection that appears valid when it is not valid. For example, at an invalid crack depth to wall thickness ratio of 1.1 and parameters set to their mean values, the probability of nondetection is 0.045. Table B–31 shows that this result is not much different than results from valid ratios.
3. Harrington, et al. (2011) describe a probability of detection curve that is nearly flat and this shape is reflected in the results shown in Table B–31. Clarify in Harrington, et al. (2011) how this shape is valid over the range of crack depth to wall thickness ratios from 0.1 to 1.0 and with a nonzero probability of nondetection at a ratio of 1.0.

Table B–30. Input and Output for In-Service Inspection Module: ISI_v2.1.f90			
ISI Module Input	Description	ISI Module Output	Description
aot (0 <aot<1)	Crack Depth to Wall Thickness Ratio	ldetected	Detection Flag
Beta1	β_1 parameter in Equation 15	PND (i.e., 1-POD)	Probability of nondetection (PND)
Beta2	β_2 parameter in Equation 15	time_end_time_begin	DLL CPU clock time
Urnd (0<urnd <1)	Uniform random number	—	—

Table B–31. In-Service Inspection Module Probability of Nondetection			
Crack Depth to Wall Thickness Ratio	Beta1	Beta2	Probability of Nondetection
0.1	2.7076	0.31	0.061
0.3	2.7076	0.31	0.057
0.4	2.7076	0.31	0.056
0.9	2.7076	0.31	0.048
1.0	2.7076	0.31	0.047

B3 CONCLUSIONS

This section summarizes the verification testing that was documented in Section B2. In Section B3.1, for conciseness the summary is presented in tabular form. Section 3.2 summarizes recommendations for resolving possible coding issues found during verification testing.

B3.1 Summary of Verification Testing

The xLPR modules that underwent verification testing are listed in Table B–1, Section 1.1. Table B–32 summarizes the verification testing.

Table B–32. Summary of Verification Testing	
Module Name	Verification Summary
Coalescence_v2.2	Verified using spreadsheet calculations and comparison with Harrington, et al. (2011)*
crack_init_v2.1	Direct Model 1: No issues Direct Model 2: Possible code error (see Table B–3), other minor code and documentation inconsistencies Weibull Model: No issues
grower_v2.1	Major coding error from Harrington, et al. (2011, Eq. 3.21)*
kSurf_v1.1	Equations spot checked with “Surface Crack Anderson.xlsx” and Anderson, et al. (2002)† Verified by comparison of output with ConceptualDescription-XLPR-DSK-004R0_SIAM_kSurf_v1.1.docx (Williams and Klasky, 2010)‡
kTWC_v1.1	Influence function data spot checked with “TWC Anderson.xlsm” Verified by comparison of output with ConceptualDescription-XLPR-DSK-004R0_SIAM_kTWC_v1.1.docx (Williams and Klasky, 2010)§
SCFail_v2.1	Equations spot checked with Rahman (1998), Brust (1987)¶, and Harrington, et al. (2011)* Verified by comparison of output with ConceptualDescription-XLPR-DSK-004R0_SIAM_SCFail_v2.1.docx (Williams, 2010)#
TWCFail_v2.1	Equations spot checked with Rahman (1998), Gilles and Brust (1991)**, Brust and Gilles (1994)††, and Brust (1987)¶¶ Verified by comparison of output with ConceptualDescription-XLPR-DSK-004R0_SIAM_TWCFail_v2.1.docx (Williams, 2010)‡‡
COD_v2.1	Equations spot checked with Olson (2010)§§ and Mattie, et al. (2010) Verified using checkCODdriver.exe and Mathematica™ and comparison with Olson (2010) §§ and Mattie (2010) ¶¶¶
ISI_v2.1	Verified using spreadsheet calculations and comparison with Harrington, et al. (2011)*
<p>*Harrington, C., F. Ammirato, B. Brust, D. Dedhia, E. Focht, M. Kirk, C. Lange, R. Olsen, P. Scott, D.J. Shim, and G. White. “Models and Inputs Selected for Use in the xLPR Pilot Study.” Product Identifier 1022528. Palo Alto, California: Electric Power Research Institute. 2011.</p> <p>†Anderson, L., G.. Thorwald, D.J. Revelle, D.A. Osage, J.L. Janelle, and M.E. Fuhry. “Development of Stress Intensity Factor Solutions for Surface and Embedded Cracks in API 579.” <i>WRC Bulletin 471</i>. Shaker Heights, Ohio: Welding Research Council. 2002.</p> <p>‡Williams, P. and H. Klasky. “ConceptualDescription-XLPR-DSK-004R0_SIAM_kSurf_v1.1.” Oak Ridge, Tennessee: Oak Ridge National Laboratory. 2010.</p> <p>§Williams, P. and H. Klasky. “ConceptualDescription-XLPR-DSK-004R0_SIAM_kTWC_v1.1.” Oak Ridge, Tennessee: Oak Ridge National Laboratory. 2010.</p> <p>¶Rahman, S. “Net-Section-Collapse Analysis of Circumferentially Cracked Cylinders—Part II: Idealized Cracks and Closed-Form Equations.” <i>Engineering Fracture Mechanics</i>. Vol. 61. pp. 213–230. 1998.</p> <p>¶¶Brust, F.W. NUREG/CR–4853, “Approximate Methods for Fracture Analysis Through-Wall Cracked Pipes.” Washington, DC: NRC. pp. 2–13. 1987.</p> <p>#Williams, P. “ConceptualDescription-XLPR-DSK-004R0_SIAM_SCFail_v2.1.” Oak Ridge, Tennessee: Oak Ridge National Laboratory. 2010.</p> <p>**Gilles, P. and F.W. Brust. “Approximate Fracture Methods for Pipes—Part I: Theory.” <i>Nuclear Engineering and Design</i>. Vol. 127. pp. 1–17. 1991.</p> <p>††Brust, F.W. and P. Gilles. “Approximate Methods for Fracture Analysis of Tubular Members Subjected to Combined Tensile and Bending Loads.” <i>Journal of Offshore Mechanics and Arctic Engineering</i>. Vol. 116. pp. 221–227. 1994.</p> <p>‡‡Williams, P. “ConceptualDescription-XLPR-DSK-004R0_SIAM_TWCFail_v2.1.” Oak Ridge, Tennessee: Oak Ridge National Laboratory. 2010b.</p> <p>§§Olson, R. “xLPR Model Document and Checking Desktop Guide, Model ID: COD_v2.0.” Columbus, Ohio: Battelle. 2010.</p> <p>Mattie, P.D., C.J. Sallaberry, J.C. Helton, and D.A. Kalinch. “Development, Analysis, and Evaluation of a Commercial Software Framework for the Study of Extremely Low Probability of Rupture (xLPR) Events at Nuclear Power Plants.” SAND2010-8480. Albuquerque, New Mexico: Sandia National Laboratories. 2010.</p> <p>¶¶¶Mattie, P.D. “xLPR Model Document and Checking Desktop Guide, Model ID: COD_v2.1.DLL.” Albuquerque, New Mexico: Sandia National Laboratories. 2010.</p>	

B3.2 Issues and Recommendations Based on Verification Testing

The following list summarizes coding issues or documentation inconsistencies that arose during verification testing. Possible recommendations to clarify these issues are also given.

1. Module crack_init_V2.1 (crack initiation), for the same temperature and stress input data, comparison of Direct 1, Direct 2, and Weibull models produced significantly different results. However, because each model produces significantly different results for the same input stress and temperature values, the appropriateness of model

parameters and validity of these different models should be reviewed and discussed in Harrington, et al. (2011). Therefore, the parameter values listed in Harrington, et al. (2011, Table 3.14) may need to be reviewed.

2. Module grower_V2.1 (crack growth), Ahluwalia (2007) shows a sample calculation pertaining to Equation 3.21 that uses the base 10 logarithm (i.e., \log_{10}) versus the natural logarithm (i.e., \ln). The code, however, incorrectly implements Harrington, et al. (2011, Eq. 3.21) by using the natural logarithm in the calculation.
3. For the module ISI_v2.1 (inservice inspection), the software logic agreed with the descriptions in Harrington, et al. (2011) and results from the software agreed with spreadsheet calculations; however, the following is recommended:
 - a. Harrington, et al. (2011) indicate that if a random sample is less than the calculated probability of detection, then the crack is considered to have been detected. Revise Harrington, et al. (2011) to indicate that if the calculated probability of nondetection is less than or equal to a randomly sampled value from a uniform distribution, then the crack is considered to have been detected.
 - b. Add error detection to the DLL. For example, if a crack depth to wall thickness ratio greater than one is input, then the DLL can still produce a probability of nondetection that appears valid when it is not valid. Additional details were provided in Section 2.4.
 - c. Harrington, et al. (2011) describe a probability of detection curve that is nearly flat, and this shape is reflected in the results shown in Table B–31. Clarify in Harrington, et al. (2011) how this shape is valid over the range of crack depth to wall thickness ratios from 0.1 to 1.0 and with a nonzero probability of nondetection at a ratio of 1.0.

B4 REFERENCES

Ahluwalia, K. "Materials Reliability Program: Mitigation of PWSCC in Nickel-Base Alloys by Optimizing Hydrogen in the Primary Water (MRP-213)." Product Identifier 1015288. Palo Alto, California: Electric Power Research Institute. 2007.

Anderson, L., G. Thorwald, D.J. Reville, D.A. Osage, J.L. Janelle, and M.E. Fuhry. "Development of Stress Intensity Factor Solutions for Surface and Embedded Cracks in API 579." *WRC Bulletin 471*. Shaker Heights, Ohio: Welding Research Council. 2002.

Battelle. "NRCPIPE User's Guide (Windows Version 3.0)." Contract NRC–04–91–063. Columbus, Ohio: Battelle. 1996.

Brust, F.W. NUREG/CR–4853, "Approximate Methods for Fracture Analysis Through-Wall Cracked Pipes." Washington, DC: NRC. pp. 2–13. 1987.

Brust, F.W. and P. Gilles. "Approximate Methods for Fracture Analysis of Tubular Members Subjected to Combined Tensile and Bending Loads." *Journal of Offshore Mechanics and Arctic Engineering*. Vol. 116. pp. 221–227. 1994.

EPRI. "Elastic-Plastic Fracture Analysis of Through-Wall and Surface Flaws in Cylinders. NP-5596." Palo Alto, California: EPRI. 1988.

Gilles, P. and F.W. Brust. "Approximate Fracture Methods for Pipes—Part I: Theory. *Nuclear Engineering and Design*. Vol. 127. pp. 1–17. 1991.

GoldSim Technology Group LLC. "GoldSim® Version 10.5." Issaquah, Washington: GoldSim Technology Group LLC. 2010.

Harrington, C., F. Ammirato, B. Brust, D. Dedhia, E. Focht, M. Kirk, C. Lange, R. Olsen, P. Scott, D.J. Shim, and G. White. "Models and Inputs Selected for Use in the xLPR Pilot Study." Product Identifier 1022528. Palo Alto, California: EPRI. 2011.

Klasky, H.B., P.T. Williams, B.R. Bass, and S. Yin. "Structural Integrity Assessments Modular-Probabilistic Fracture Mechanics (SIAM-PFM): User's Guide for xLPR." ORNL/NRC/LTR-247. Oak Ridge, Tennessee: Oak Ridge National Laboratory. 2010.

Lahey Computer Systems. "Lahey/Fujitsu Fortran v7.1 for Windows." Incline Village, Nevada: Lahey Computer Systems, Inc. 2004.

Mattie, P.D. "xLPR Model Document and Checking Desktop Guide, Model ID: COD_v2.1.DLL." Albuquerque, New Mexico: Sandia National Laboratories. 2010.

Mattie, P.D., C.J. Sallaberry, J.C. Helton, and D.A. Kalinch. "Development, Analysis, and Evaluation of a Commercial Software Framework for the Study of Extremely Low Probability of Rupture (xLPR) Events at Nuclear Power Plants." SAND2010-8480. Albuquerque, New Mexico: Sandia National Laboratories. 2010.

Microsoft Corporation. "Microsoft Office Excel® 2007." Redmond, Washington: Microsoft Corporation. 2007.

Olson, R. "xLPR Model Document and Checking Desktop Guide, Model ID: COD_v2.0." Columbus, Ohio: Battelle. 2010.

Press, W.H., W.T. Vetterling, S.A. Teukolsky, and B.P. Flannery. *Numerical Recipes, The Art of Scientific Computing, Fortran 77*. Cambridge, Massachusetts: Cambridge University Press. 1992.

Rahman, S. "Net-Section-Collapse Analysis of Circumferentially Cracked Cylinders—Part II: Idealized Cracks and Closed-Form Equations." *Engineering Fracture Mechanics*. Vol. 61. pp. 213–230. 1998.

U.S. Nuclear Regulatory Commission and Electric Power Research Institute. "xLPR Version 1.0 Report—Technical Basis and Pilot Study Problem Results." Washington, DC: U.S. Nuclear Regulatory Commission, Office of Nuclear Regulatory Research; Palo Alto, California: EPRI. 2011.

Williams, P. "Conceptual Description-XLPR-DSK-004R0_SIAM_SCFail_v2.1." Oak Ridge, Tennessee: Oak Ridge National Laboratory. 2010a.

Williams, P. "Conceptual Description-XLPR-DSK-004R0_SIAM_TWCFail_v2.1." Oak Ridge, Tennessee: Oak Ridge National Laboratory. 2010b.

Williams, P. and H. Klasky. "Conceptual Description-XLPR-DSK-004R0_SIAM_kSurf_v1.1." Oak Ridge, Tennessee: Oak Ridge National Laboratory. 2010a.

Williams, P. and H. Klasky. "Conceptual Description-XLPR-DSK-004R0_SIAM_kTWC_v1.1." Oak Ridge, Tennessee: Oak Ridge National Laboratory. 2010b.

Wolfram Research, Inc. "Wolfram Mathematica™ 7.0." Champaign, Illinois: Wolfram Research, Inc. 2008.

APPENDIX C

IMPLEMENTATION OF A “DUMMY” MODULE IN XLPR-GoldSim AND XLPR-SIAM

The objective of this appendix is to document efforts that have been made to simulate development and implementation of a “new” test module into both the xLPR-GoldSim (XLPR-GoldSim) and xLPR-SIAM (XLPR-SIAM) platforms. This has been done as a test exercise to gain insight into analyses of the flexibility of the frames to incorporate updates and perform code maintenance. This work follows the general requirements that are described in Chapter 4 of this report. An abbreviated description of the requirements for the test module follows.

For this exercise, the test module had the following requirements:

- The module should be developed in FORTRAN and compiled into a dynamically linked library (DLL) for use with the GoldSim-based XLPR-GoldSim or with the F2Py utility for use with the Python-based XLPR-SIAM.
- The module should take as input two random parameters, r_1 and r_2 , and should apply these random factors to the crack depth and crack length (the depth and length should also be provided as inputs to the procedure).
- The parameters r_1 and r_2 should be defined as random parameters where
 - The inputs for these random parameters are in the master spreadsheet (in the case of XLPR-GoldSim) or in one of the input parameter tabs (in the case of XLPR-SIAM)
 - The random parameters can be specified as distributions in xLPR (e.g., normal, uniform, log distributions)
 - The parameters can be switched from aleatory uncertainty to epistemic uncertainty, and from one distribution to another
- The random parameters r_1 and r_2 should be applied after the subroutine grower (grower_v2.1.f90). More specifically, the factors should be applied to all active cracks. If applicable, the physical bounds for the system (e.g., thickness or diameter) should be checked to ensure they are not exceeded due to integration of the module.
- The parameters r_1 and r_2 should be tracked in the set of appropriate random input parameters (epistemic or aleatory) to support sensitivity analyses. Further, these new input parameters should be added to appropriate elements in GoldSim output files or databases.

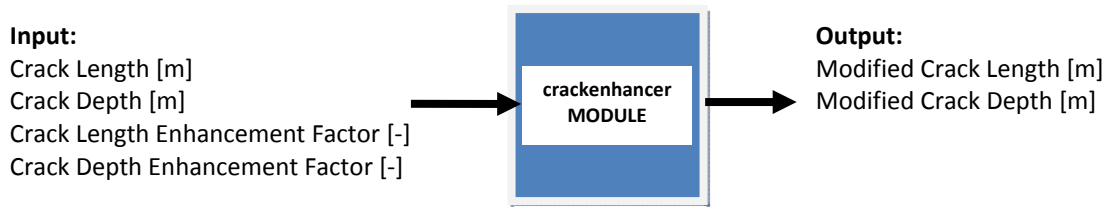
Also, during the development and incorporation of the test module, the following information will be used to compare the model frameworks (the results of this comparison are reflected in Table 4.1 of this report).

1. The estimated time needed to incorporate a new module to each of the codes and major difficulties faced in incorporating the new module

2. An evaluation of the availability of existing functions in frame to support the Monte Carlo frame (i.e., define input parameters, sample input parameters, record input parameters, manage data output from multiple realizations, display Monte Carlo outputs, communicate with existing modules)
3. An evaluation of the capability of the codes to expose (or display) intermediate outputs that are not in default outputs
4. An estimate of the time needed to sufficiently learn Python and GoldSim so codes can be maintained

The “New” Test Module

The test module that will be implemented will be a FORTRAN-based subroutine, crackenhancer_v0.1.f90. The module will receive an input array and an output array. The input array should pass the crack length and the crack depth, as well as the random enhancement factors r_1 and r_2 , which will be referred to as the cracklengthenhancement and crackdepthenhancement factors, respectively. The module will simply multiply the crack length and the crack depth by their respective enhancement factors, which are unitless, and then return the modified values for the crack length and depth (i.e., as depicted in the following simplified diagram):



For interaction with GoldSim, the module must be compiled as a DLL. Also, the file must have additional input/output values [for details, see GoldSim Technology Group (2011, Appendix C)].

Implementing the CrackEnhancer Module Into the XLPR-GoldSim Framework

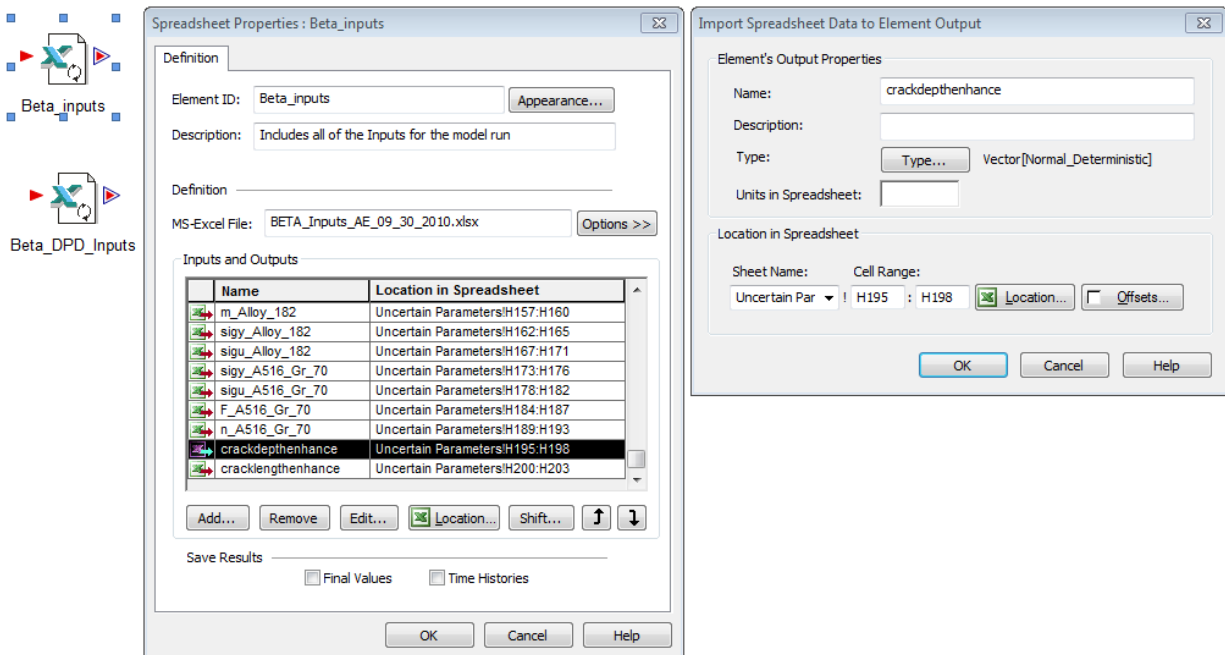
The steps taken to implement the module in the XLPR-GoldSim framework are detailed next.

1. First, it was necessary to add the new crack enhancement parameters to the framework. To do this, the model implementation requires several substeps.
 - a. The new parameters were incorporated into the framework by first adding the uncertain (random) parameters in the BETA_Inputs_AE_09_30_2010.xlsx Microsoft® Excel® spreadsheet the model used. The parameters were added to the “Uncertain Parameters” spreadsheet and are shown in the following screenshot:

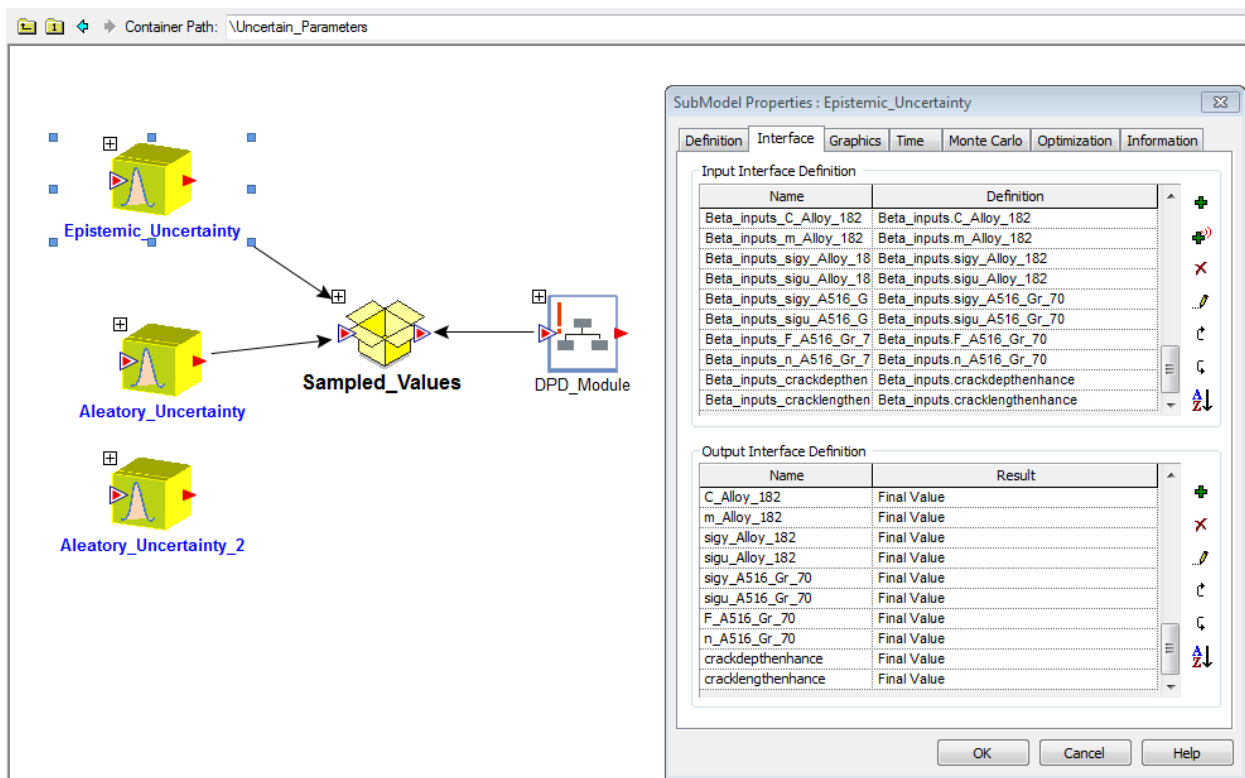
194						Distribution Type	Normal		
195						Mean	1000		
196		crackdepthenhance				Stdev	0.001		
197						Determin	1		
198	37					Type	1		
199						Distribution Type	Normal		
200						Mean	1000		
201		cracklengthenhance				Stdev	0.001		
202						Determin	1		
203	38					Type	1		

Addition of the parameters to the spreadsheet allows for persistence of input values, and manually changing the “Type” specification allows the user to change the type to either “Aleatory” or “Epistemic.” The model is written such that the parameters are sampled as both aleatory and epistemic (object elements for these must be added; this will be covered in a later step), and the type is checked in an additional switch [this logic must be added as well (type = 1 Aleatory, otherwise the element is assumed to be Epistemic)].

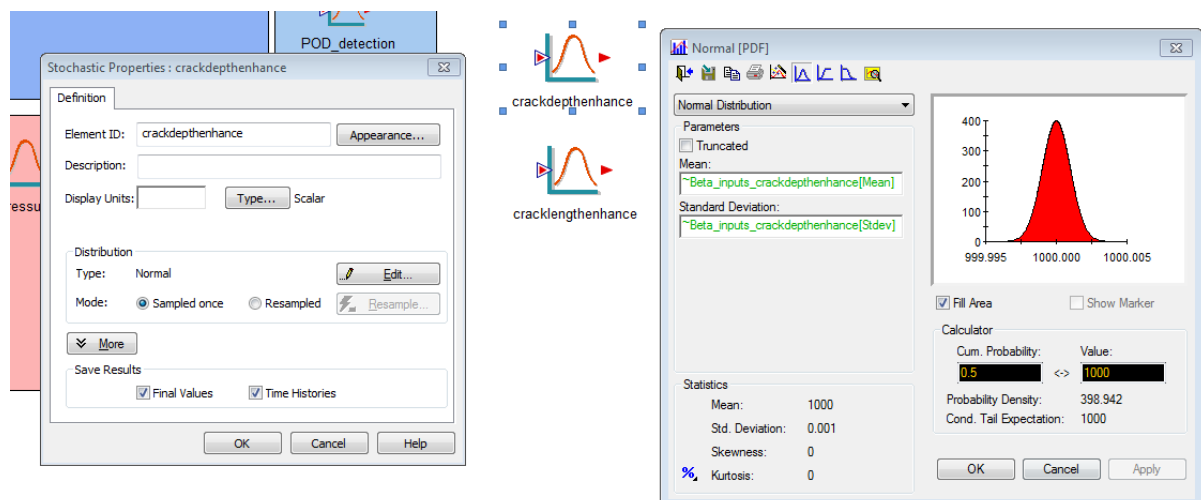
- b. However, before the parameters could be read by GoldSim, the Excel spreadsheet interface element in the XLPR-GoldSim model (“Beta_inputs” in \Data_Source) had to be modified to point to the newly added cells. An example of the required changes are shown in the following screenshot (the same procedure should be followed for the other new parameter):



- c. Next, it was necessary to add stochastic input parameters to use the array input values for the parameters that are imported at run time from the linked Excel spreadsheet. The XLPR-GoldSim model has been implemented such that there are two stochastic elements sampled for each parameter regardless of intended type (epistemic or aleatory). The locations for the sampling of the epistemic and aleatory elements are submodel elements. Because the sampled stochastic input elements are sampled in submodel elements, the submodel elements (\Uncertain_Parameters\Epistemic_Uncertainty and \Uncertain_Parameters\Aleatory_Uncertainty) must have the elements added into their interface properties; otherwise, the values imported from Excel will not be exposed to the submodel and the sampled values in the submodel (to be added) will not be exposed to the global or master models. This is done by accessing the Epistemic and Aleatory submodels and setting the interfaces (for both submodels) to pass the variables (the output interface should be done after adding the stochastic input elements to the submodel elements):

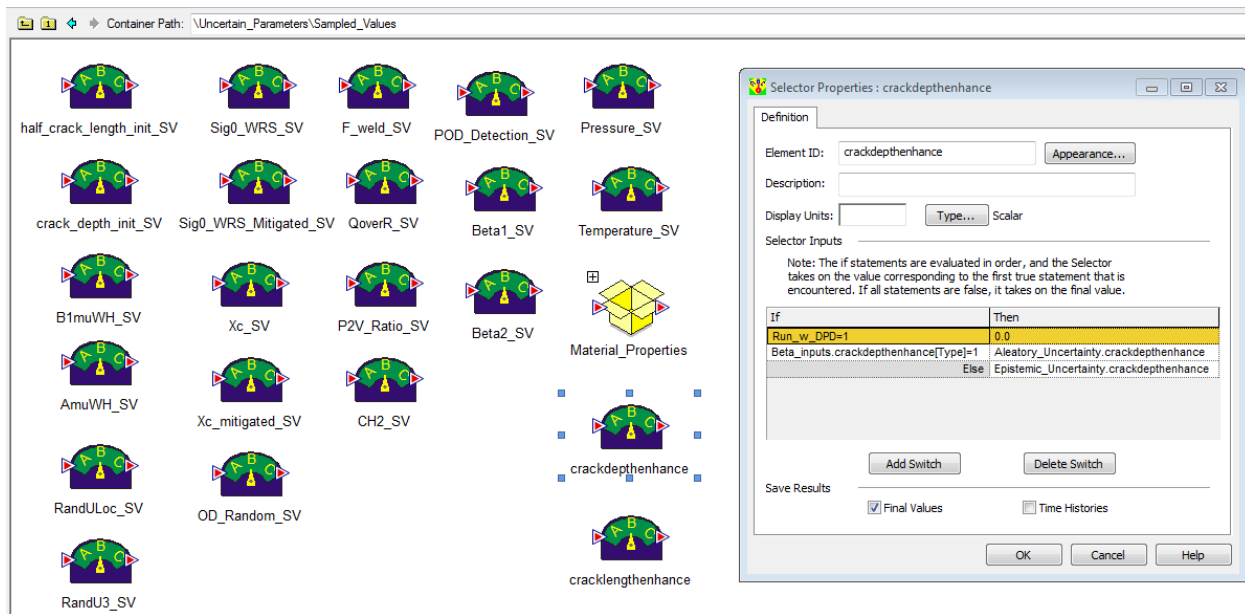


For both crack enhancement parameters, stochastic input elements were created in the \Uncertain_Parameters\Epistemic_Uncertainty\ submodel and the \Uncertain_Parameters\Aleatory_Uncertainty\ submodel. An example screenshot for the Epistemic_Uncertainty\ submodel follows:



- d. At run time, the model imports values from Excel and then, at time zero, the Epistemic and Aleatory submodels sample values for the stochastic elements contained there. To determine which sampled parameter (Epistemic or Aleatory) is used for a given parameter, the "Type" entry and the GoldSim selector elements (which are located in \Uncertain_Parameters\Sampled_Values\) are

specified in the Excel spreadsheet. The next step is to add selector elements for both crack enhancement factors; this is shown in the following screenshot.



The DPD option is a model capability that is not required for the current release of the xLPR, and this capability is not evaluated in this testing effort.

2. The next step in implementing the crackenhancer module is to select the appropriate location to add an interface from the GoldSim model to the external FORTRAN module (compiled as a DLL). The intent is to apply the crack enhancement factors to active cracks. This can be one of the more time-consuming steps because the code must be carefully analyzed and understood; its complexity can vary widely depending upon the intended function of the module to be implemented. Additionally, although the GoldSim program offers many useful tools to navigate constructed models, the relationships between parameter values that are updated can become complex. These relationships must be examined carefully to prevent an unintended effect upon the data. The proper sequence of parameter updating should be maintained.

It was determined that the location to interface and apply the module would be in the crack growth submodel (\Crack_Growth \Crack_Growth_Submodels), immediately following the call to the external subroutine grower (grower_v2.1.f90, grower_DLL_v2.1.dllx). The call to the external subroutine grower is made inside of the Crack_Growth_Submodels, \Crack_Growth_sc container. The crackenhancer module will be implemented in the Crack_Growth_Submodels, \Update_Crack_Status container. This container is updated (for surface cracks, Type = -1) following the update of the parameters in the Crack_Growth_sc container. Also note the Crack_Growth_Submodel is looped over all active cracks; therefore, the crackenhancer module will be applied on an individual crack basis (not to an array of crack length and depth values).

The call to the crackenhancer module is made before the parameters contained in the \Update_Crack_Status container are updated. Thus, the approach used in implementing the module is to intercept the crack length and crack depth values that have been passed through the grower module and modify the individual crack and depth values

before passing them onto the appropriate parameters (and/or functions) to update the crack array.

3. The next step is to carry out the implementation strategy described in Step 2.
 - a. Here, it is first necessary to adjust the \Crack_Growth\Crack_Growth_Submodels interface option to expose parameters from the main model to this submodel. This step is depicted in the following screenshot.

The screenshot shows the 'Crack Growth Submodels' interface within a container path of '\Crack_Growth'. The interface includes a diagram with a 'G S M' block, a 'sig0' input, a 'sigma5' input, and a 'Crack_Growth_Results' output. A yellow box contains the following text:

Crack Growth Submodels

Crack_Growth_Submodels contains the model estimating Stress and crack growth for both Surface and Through Wall Cracks. It is a submodel and thus runs locally

Crack_Growth_Results save the results of the DLLs

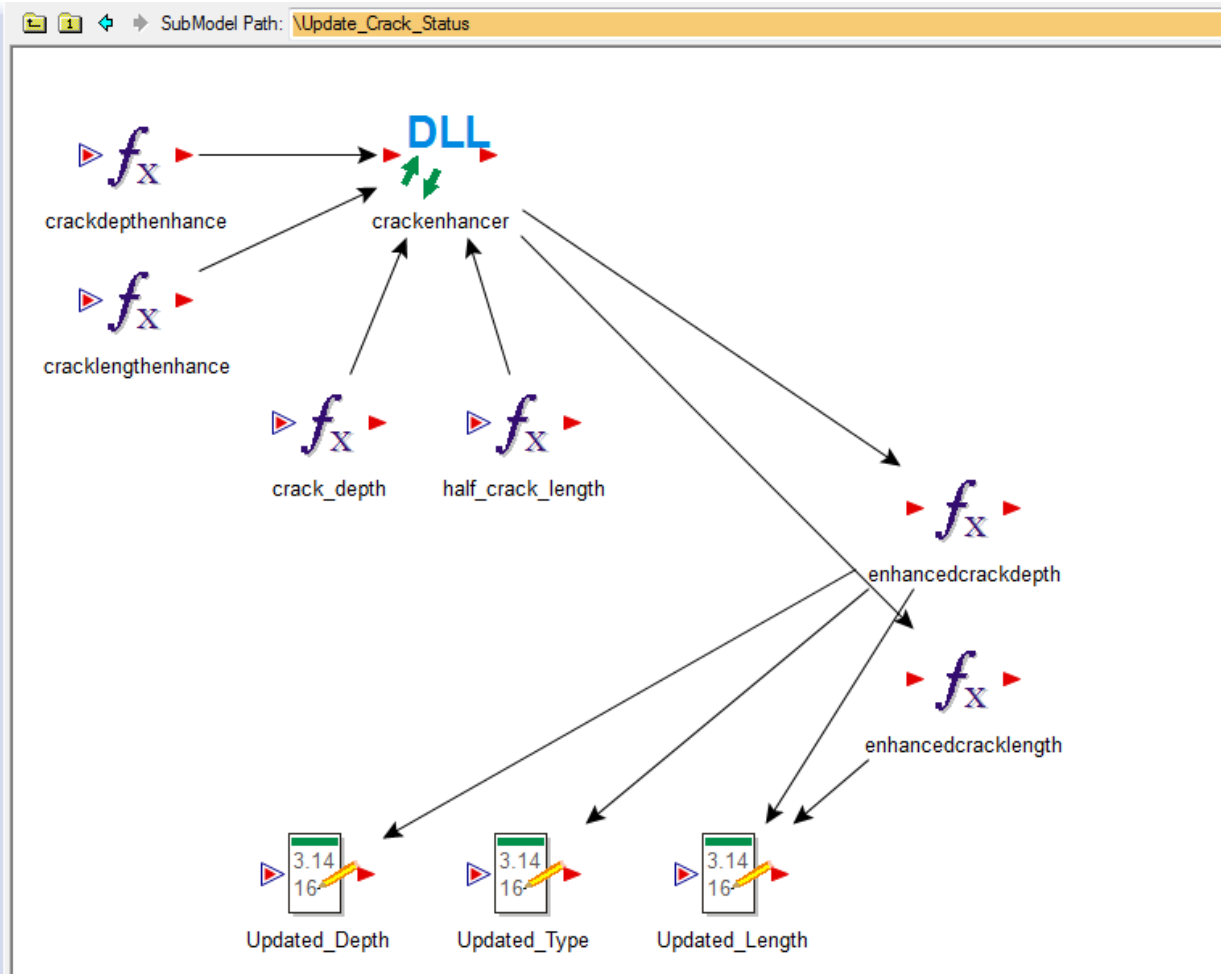
The 'SubModel Properties : Crack_Growth_Submodels' dialog is open, showing the 'Interface' tab. The 'Input Interface Definition' table is as follows:

Name	Definition
PWSCC_alpha	PWSCC_alpha
timestep_in_yr	Timestep_Length
sig0_wrs	sig0_wrs
H2	H2
Zinc	Zinc
P2V_Ratio	P2V_Ratio
CH2	CH2
IDBG	Grower_IDBG
crackdepthenhance	crackdepthenhance
cracklengthenhance	cracklengthenhance

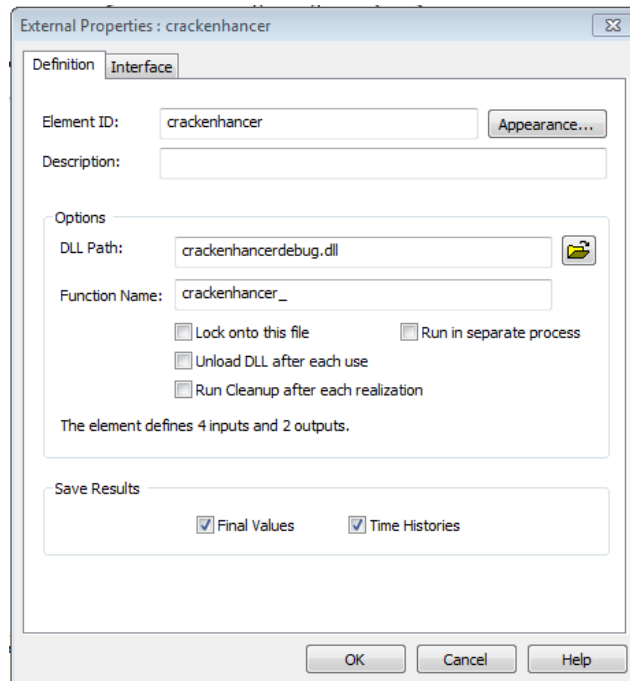
The 'Output Interface Definition' table is as follows:

Name	Result
Ksurf0	Final Value
Ksurf90	Final Value
Ktwc	Final Value
Updated_Depth	Final Value
Updated_Length	Final Value
Updated_Type	Final Value
Ksurf_DLL_Runtime	Final Value
Ktwc_DLL_Runtime	Final Value
Grower_sc_Ierr_grw	Final Value
Grower_twc_IErr_grw	Final Value

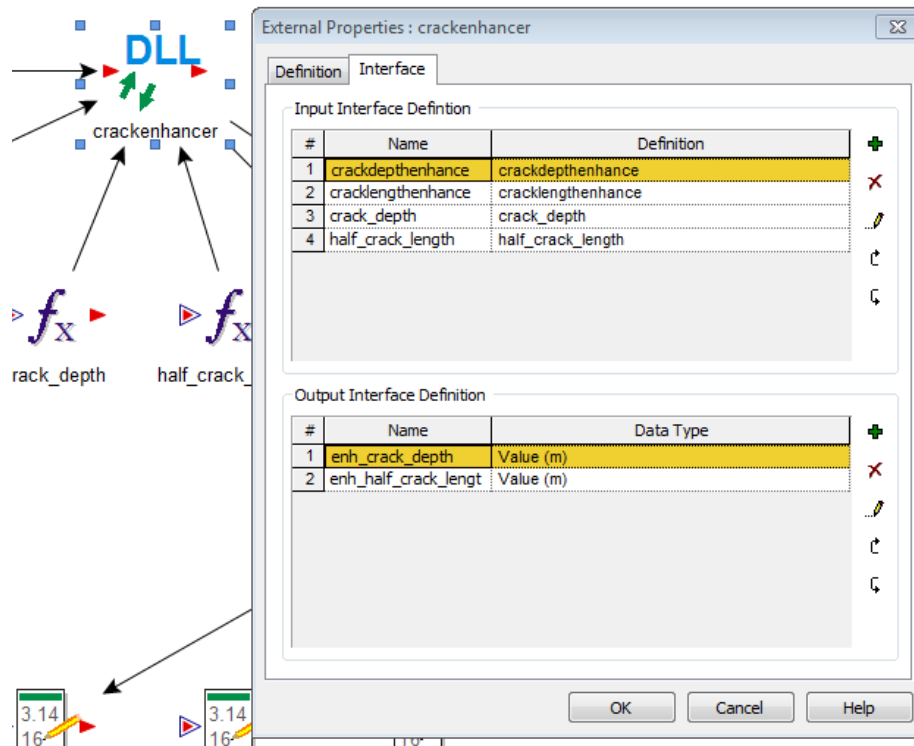
- b. The next step is to add parameters necessary to pass input into the crackenhancer module, the external function/DLL link to the crackenhancer DLL, and parameters to handle output from the crackenhancer module. To do this, several function expressions were added to the Crack_Growth\Crack_Growth_Submodels, \Update_Crack_Status container. The expression objects, along with the external function/DLL links to crackenhancer, are shown in the following screenshot.



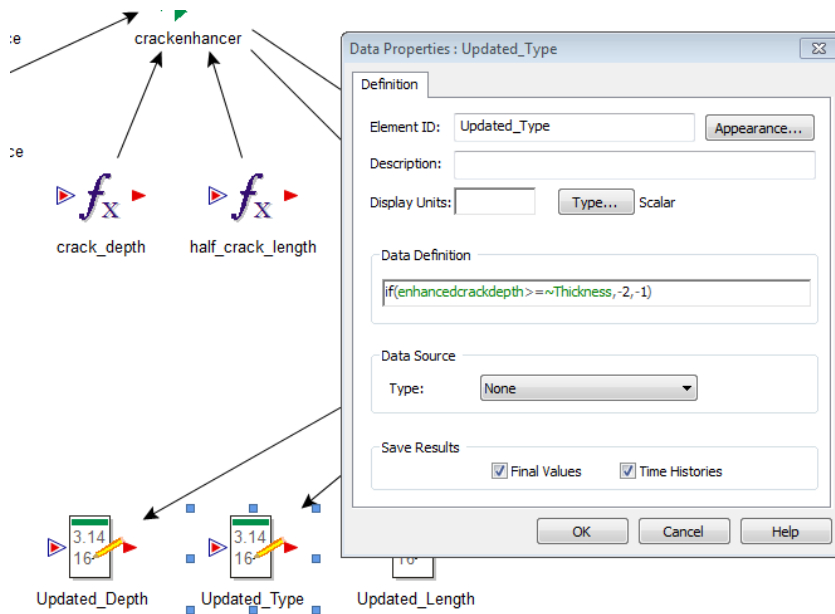
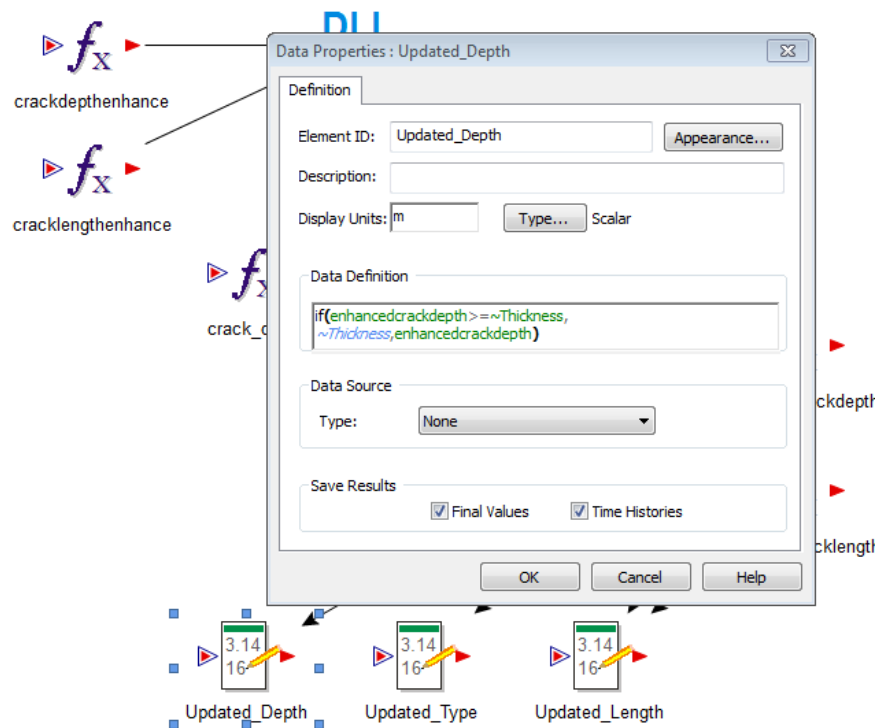
The parameters crackdepthenhance and cracklengthenhance point to the current sampled values for the crack length and depth enhancement parameters. The crack_depth and half_crack_length parameters point to the respective values as returned from the grower module (i.e., `Grower_sc.crack_depth` and `Grower_sc.half_crack_length`). The parameters enhancedcrackdepth and enhancedcracklength are the modified values returned from the crackenhancer module. Note that for the DLL object, the object must be linked to the DLL file (the details of building the DLL file will be covered in a subsequent step) and the interface options must be selected such that the input parameters and the output parameters are properly passed (these must be in the correct order for the module input and output arrays). The settings for the DLL element are shown in the following two screenshots.



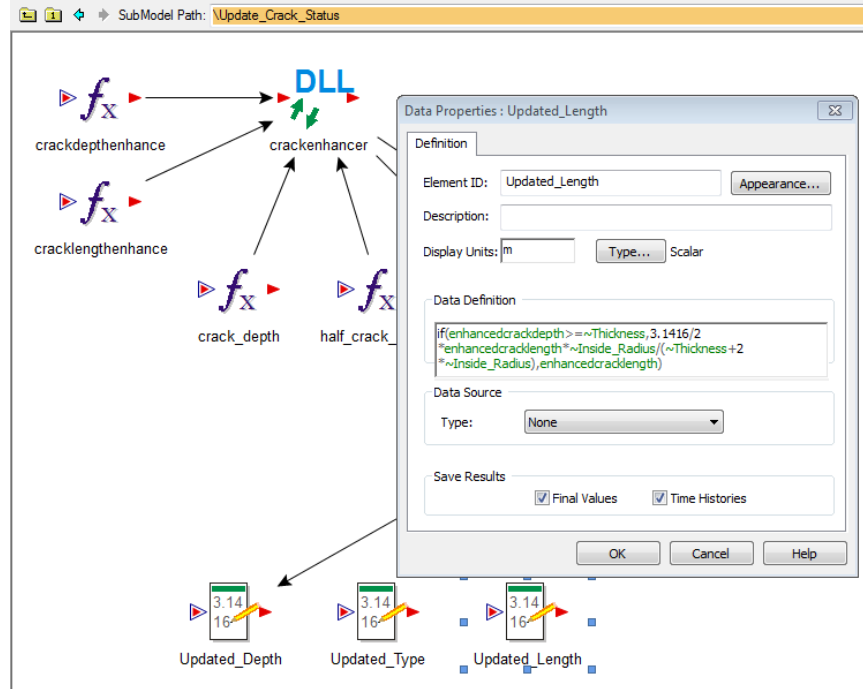
Update_Crack_Status



- c. In addition to the elements added to the \Update_Crack_Status container, the definitions for the parameters Updated_Depth, Updated_Type and Updated_Length must be updated so that they use the enhanced crack depth and size values. The appropriate changes are shown in the following screenshots:



For the Updated_Type, if the crack length is greater than or equal to the wall thickness, the crack type is changed to -2, indicating a through wall crack.



4. Additionally, the crackenhancer module must be compiled as a DLL and have the appropriate elements to interact with GoldSim [for details, see GoldSim Technology Group (2011, Appendix C)].
 - a. Based upon requirements from GoldSim, the DLL module must have additional input parameters (i.e., a method and a state variable), which are used for controlling calls to the DLL (e.g., initializing the DLL, querying number of input/output variables). This requirement thus makes it necessary to add the appropriate logic to the subroutine to handle this communication requirement.
 - b. The module source code is relatively simple and is included next for reference.

```
! This is a simple add-on module to test the complexity of adding an
! additional module to XLPR-SIAM and XLPR-GoldSim. The purpose of the module is to
! multiply crack length and crack depth values by the respective
! enhancement factors.
! -----
```

```
Subroutine crackenhancer(method, state, input, output)
  implicit none

  INTEGER(4), value      :: method ! required by GoldSim(TM)
  INTEGER(4)             :: state ! required by GoldSim(TM)
  REAL(8)                :: input(*), output(*)

  real(8) :: retavalue = 0 ! temp. holder for enhanced depth
  real(8) :: retcvalue = 0 ! temp. holder for enhanced length

  real(8) :: cracklengthenhance = 0 ! length enhancement factor
  real(8) :: crackdepthenhance = 0 ! depth enhancement factor

  real(8) :: crackdepth = 0 ! crack depth [m]
  real(8) :: crackhlfen = 0 ! crack half-length [m]

  real(8), dimension(2) :: outvector ! variable for output data

  CHARACTER(len=22) :: logfilename = 'testlogger.log' ! log filename
```



```

INTEGER :: ioerror

! The case options are following requirements from GoldSim(TM).
! See the User's Manual Appendix C for detailed information.
select case (method)

  case(0) ! Initialize
    continue
  case(1) ! Perform calculations

    crackdepthenhance = input(2)
    cracklengthenhance = input(1)
    crackdepth = input(3)
    crackhlflen = input(4)

    ! Simply multiply the crack depth and half-length by their
    ! respective enhancement factors
    retavalue = crackdepthenhance*crackdepth
    retcvalue = cracklengthenhance*crackhlflen

    output(1) = retavalue
    output(2) = retcvalue

    !   ELT, 12/27/2010 --- Opening log file to for intermediate values
    !   Open the input WRITE(*,*) fulldataset(ivars,icases)file
    OPEN ( UNIT=99, FILE=logfilename, STATUS='OLD', &
      ACTION='WRITE', ACCESS='APPEND', IOSTAT=ioerror )
    WRITE(99,*) '*****'
    WRITE(99,*) 'input array values:'
    WRITE(99,*) 'input(1) = ',input(1)
    WRITE(99,*) 'input(2) = ',input(2)
    WRITE(99,*) 'input(3) = ',input(3)
    WRITE(99,*) 'input(4) = ',input(4)
    WRITE(99,*) 'output array values:'
    WRITE(99,*) 'ouput(1) = ',output(1)
    WRITE(99,*) 'ouput(2) = ',output(2)
    WRITE(99,*) '.....'

  case(2) ! Report version number.
    output(1)=0.1
  case(3) ! Report input/output number array arguments
    output(1)=4.00 ! Number of input variables in input array
    output(2)=2.00 ! Number of output variables in output array
  case(99) ! Clean up
    continue
end select

end subroutine crackenhancer

```

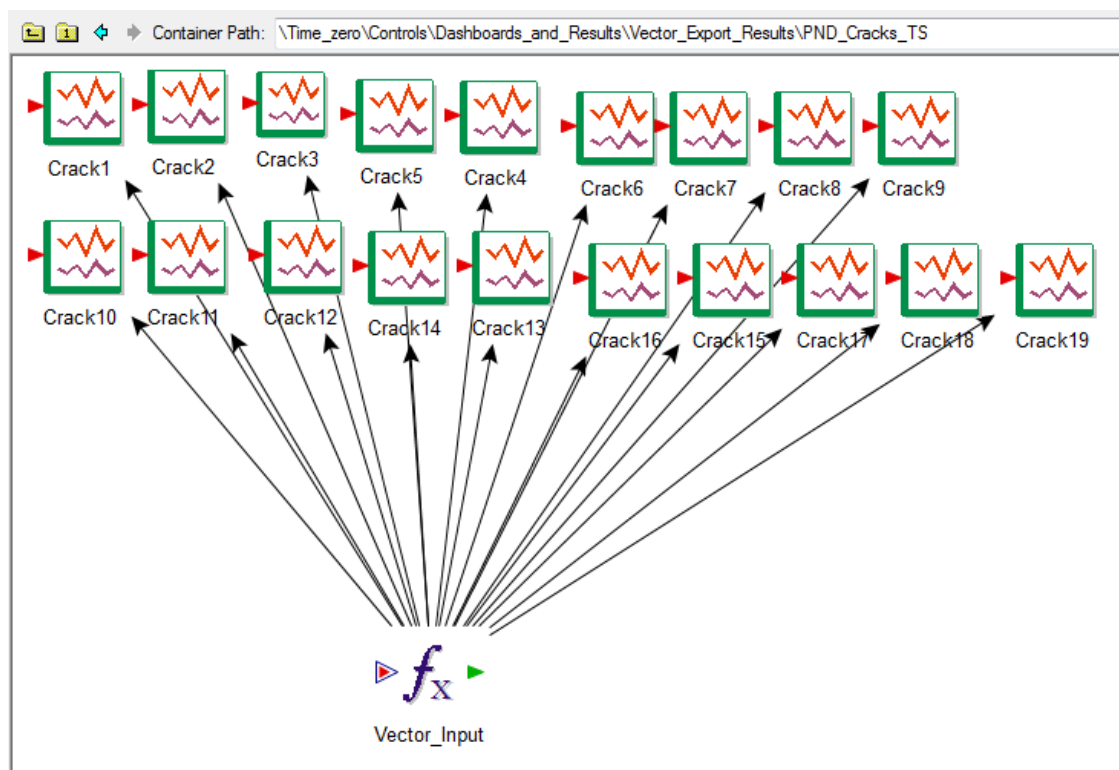
- c. A separate write to an external file was added to the model (see previous source) which logs the module input and output array values. This was done to verify that the module was functioning as expected.
- d. The authors of the XLPR-GoldSim utilized the Intel® FORTRAN compiler. In the current effort, the crackenhancer module was implemented using the GNU FORTRAN compiler, gfortran. The following commands (and flags) were used in building the DLL:

```
gfortran -shared -mrtd -o <Name of DLL File> <Object Code>
```

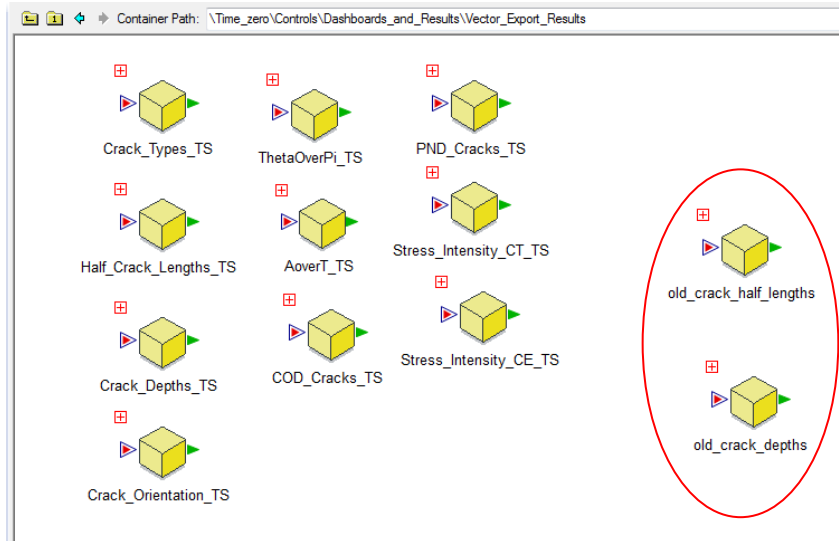
In the source code, specification of the exposed function/subroutine is not needed if using gfortran. Other compilers differ in this regard.

5. To fully assess the flexibility of the frameworks for exposing of parameter values for output and post-processing, an additional task was performed to expose the crack half-length and depth values before modification by the “new” crack enhancer module to external logging and post-processing. The process is detailed as follows.

The results exported for post-processing in XLPR-GoldSim are located in \Time_zero\Controls\Dashboards_and_Results\Vector_Export_Results. Within this container, there are several local containers, and each contain an expression (called Vector_input) linked to several time history elements, one for each of the 19 tracked cracks (see the following screenshot). Each of the time history elements contains an option to export results to an external text file. “Export automatically when simulation completes” should be selected even though the global “Automatic Export for Result Elements” (see Model, Options tab from the main GoldSim window) may be set to not export results. If “Export automatically when simulation completes” is not selected, the results will not export even if the “Export Now” command is requested from the Model, Options tab.

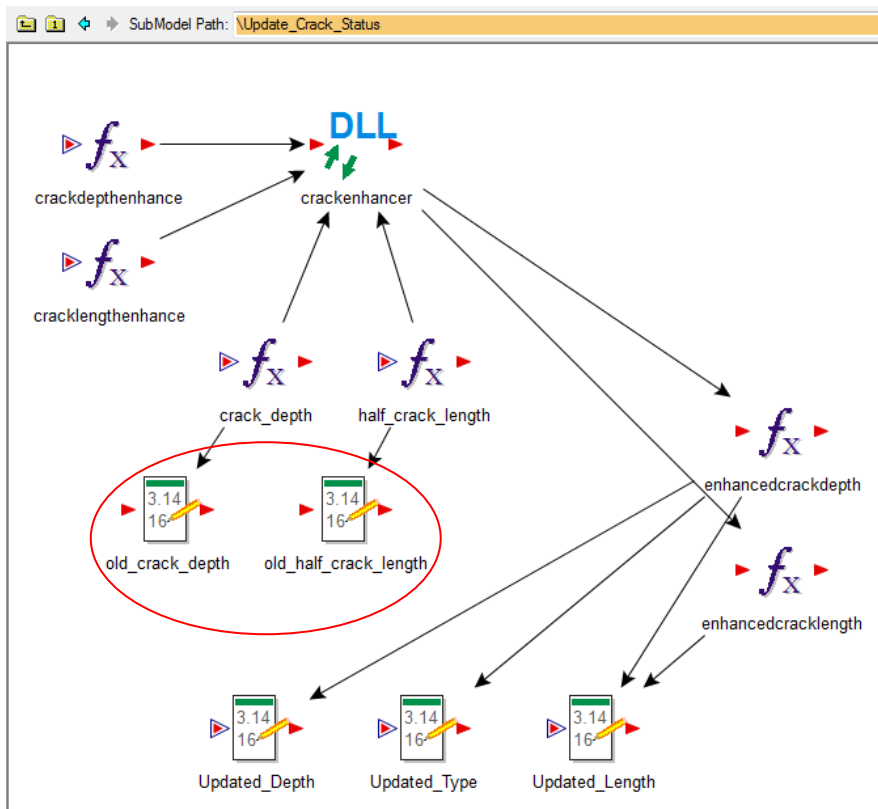


Localized containers were created in the \Time_zero\Controls\Dashboards_and_Results\Vector_Export_Results folder so that the desired value could be linked to the new Vector_Input expression elements, one for each container. Another container (i.e., PND_Crack_TS) was copied, pasted, and renamed. The new localized containers, old_crack_half_lengths and old_crack_depths, are shown in the following screenshot.

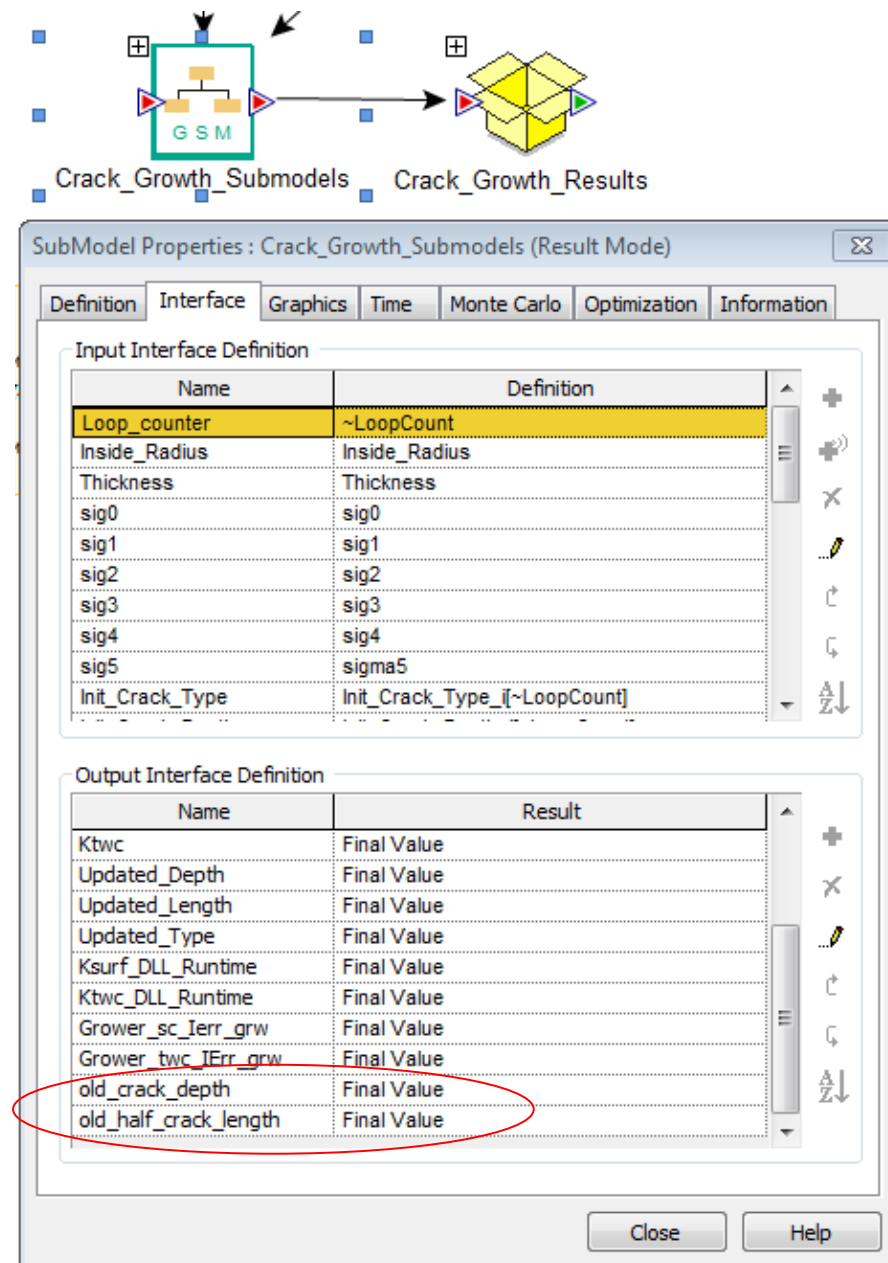


The options for the Vector_Input expression and each of the Crack# time history elements (i.e., Crack1, Crack2 and so on) were edited in the new containers. It is also necessary ensure the unmodified crack length and depth values can be passed to the respective Vector_Input expressions. This requires several steps.

First, for convenience (naming of variables), new data elements were created in the \\Crack_Growth\\Crack_Growth_Submodels\\Update_Crack_Status folder. These parameters were added to pass the crack depth and half-length values before the values are passed to the crackenhancer module.

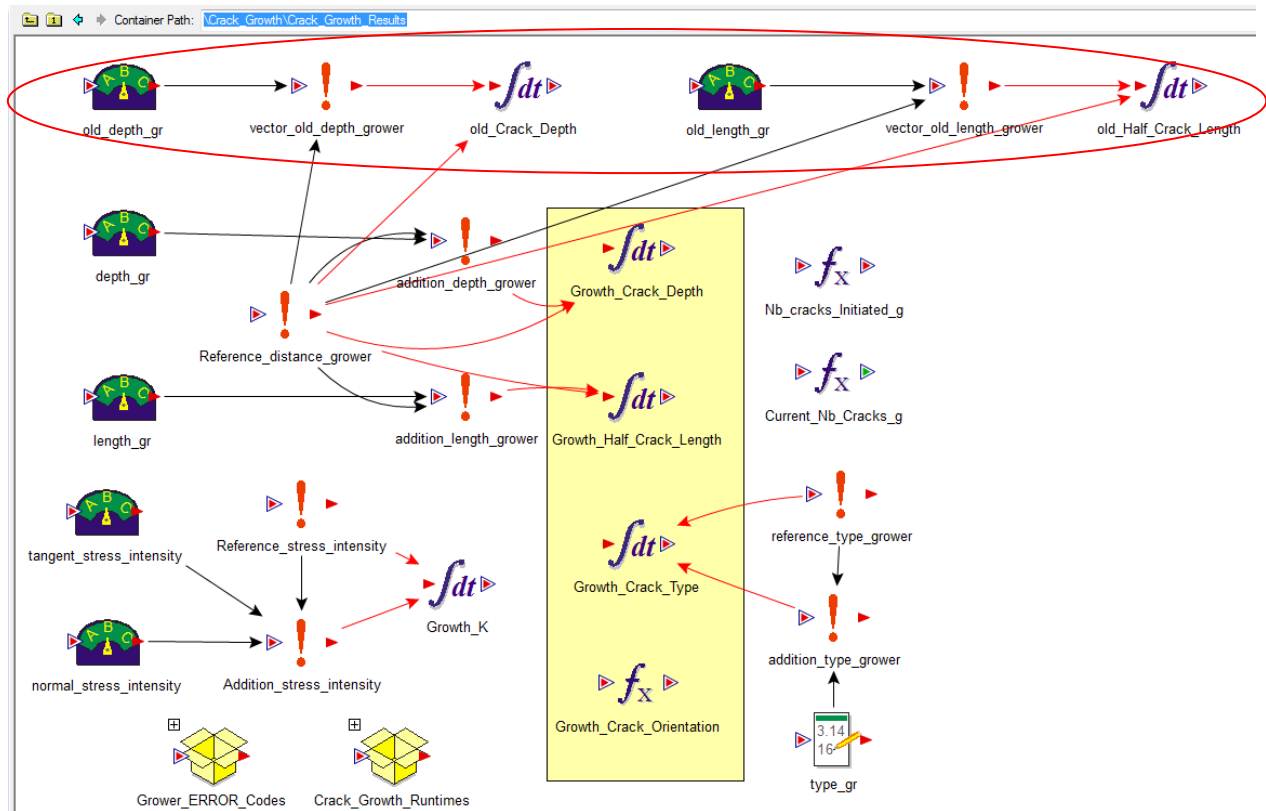


The old_crack_depth and old_half_crack_length values were exposed outside of the local Crack_Growth_Submodel. To do this, the values were added to the Output Interface Definition list in the Crack_Growth_Submodel properties as shown in the following screenshot.

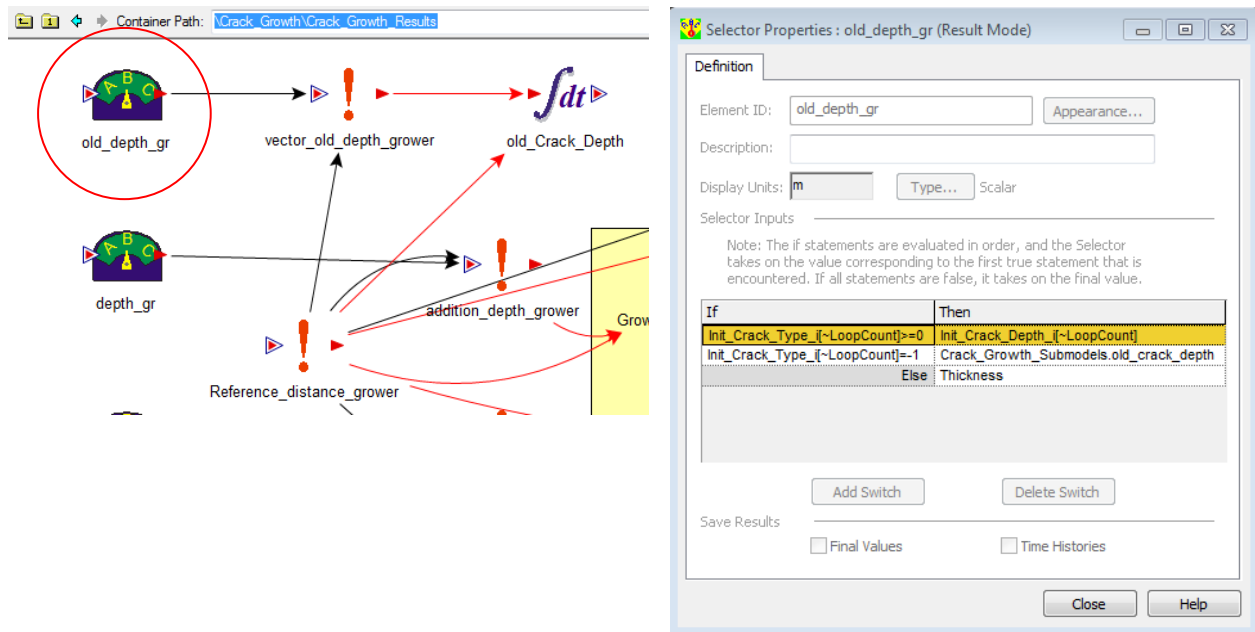


The crack length and depth values at this point are only for individual cracks, which are values for a given realization and timestep. Therefore, the values need to be collected into vectors containing the 19 cracks being tracked, which are updated for the given timestep/realization. This step is done for several parameters in the \Crack_Growth\Crack_Growth_Results container, which is also depicted in the following screenshot. To accumulate the individual crack values for the 19 tracked cracks, the XLPR-GoldSim model authors utilized several elements (selector elements and discrete change elements with triggers) to update an

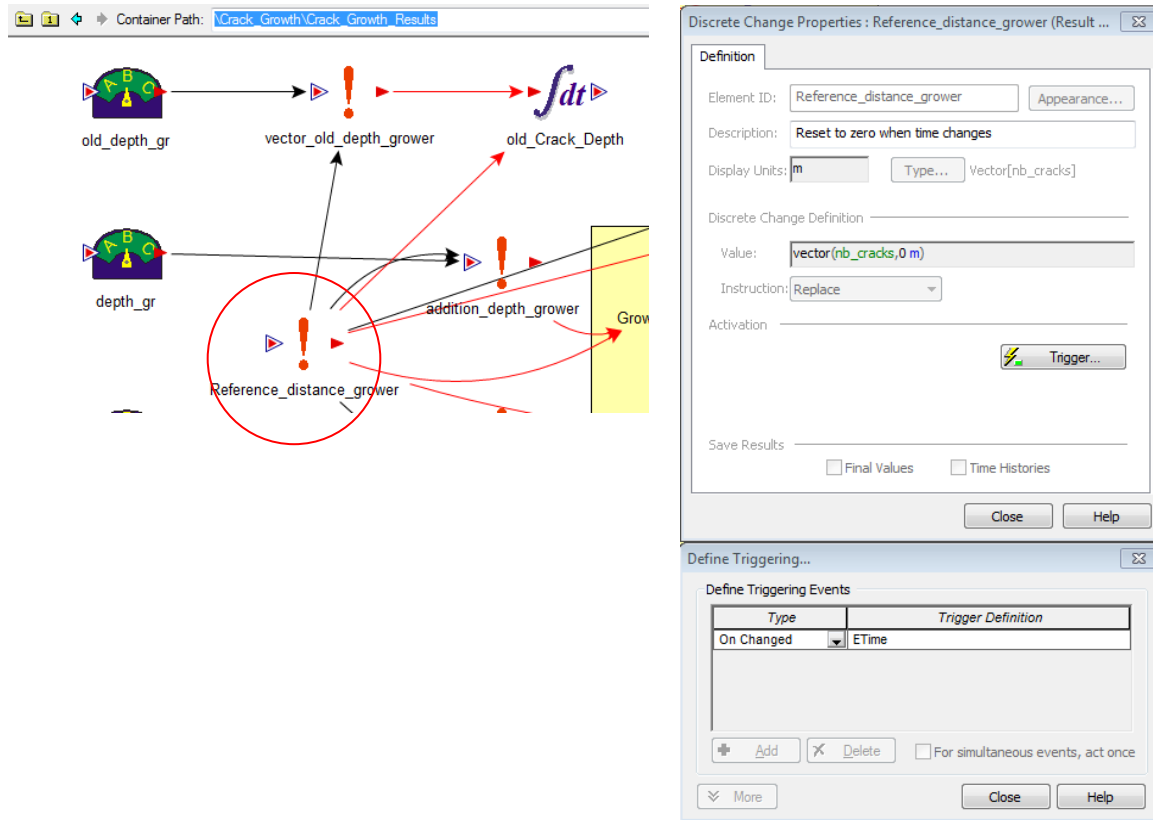
expression vector containing 19 array elements representing each crack value for the desired parameter.



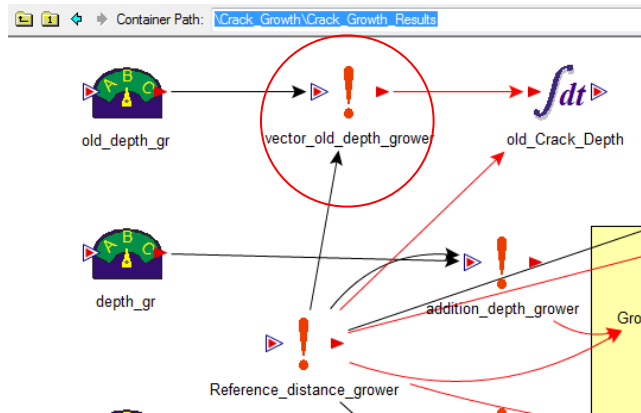
An example for the crack depth is shown in the following screenshots. First, a Selector element, “old_depth_gr,” was added to ensure the initial crack depth is updated when the crack type is -1 for a surface crack (tracked by the LoopCount, which loops over the 19 respective cracks). In this case, the updated value is linked to the old_crack_depth value passed from Crack_GrowthSubmodels.



Here, two Discrete Change elements are used in conjunction with an Integrator element; the Discrete Change elements are used to trigger updating of the Integrator element. The Discrete Change element “vector_old_depth_grower” is used to generate a vector of 19 elements (each representing a crack). The elements are all zero with the exception of the element corresponding to the present loop count (i.e., the loop count is used to track the crack index). The intent is to generate a vector of zeros, except in one entry tracking the property of the updated crack. The vector (“vector_old_depth_grower”) is then added to another vector (i.e., the Integrator element, “old_Crack_Depth”) keeping track of the 19 cracks. The sequential updating produces a vector containing the 19 crack depths for the given timestep. In setting up this sequence of parameter updating, note there are several parameter options for the Discrete Change and Integrator elements. As shown in the following screenshot, a Discrete Change element, “Reference_distance_grower,” is triggered (updated) on a change in the timestep (etime) and is set to initialize a vector containing 19 elements, all zero. Thus, the value is calculated (initialized) at the beginning of each timestep. This element is used in the options/calculation of the remaining two elements, the Discrete Change element “vector_old_depth_grower” and the Integrator element “old_Crack_Depth.” The Instruction field is set to “Replace,” and when this event is triggered for a new timestep, the Integrator is reset.



The Discrete Change element “vector_old_depth_grower” is the vector containing all zeros except for the entry corresponding to the tracked crack by the loop count. See the following screenshot for the options of the Value and Trigger settings. The Value entry in Discrete Change Properties is specified as `vector(nb_cracks, if(row=~LoopCount, old_depth_gr, 0 m * Reference_distance_grower.Follow_this))`. This specifies that all of the elements are zero except the current crack, which has the value from the Selector old_depth_gr. The “Follow_this” command used in the value entry is a special GoldSim command that is used to control the calculation sequence (GoldSim Technology Group, 2011, p. 304). This value ensures that the value is updated after the initialization of the “Reference_distance_grower” in the current timestep. The Trigger setting ensures that the present element is updated during each pass through the looping over cracks. The value for Instruction is set to “Add” to force an update of the Integrator element when the Integrator element vector is added to the present vector from the Discrete Change element, thus updating the crack depth for the present crack/loop iteration.



Discrete Change Properties : vector_old_depth_grower (Result M...

Definition

Element ID: Appearance...

Description:

Display Units: Type... Vector[nb_cracks]

Discrete Change Definition

Value:

Instruction:

Activation

Save Results ☐ Final Values ☐ Time Histories

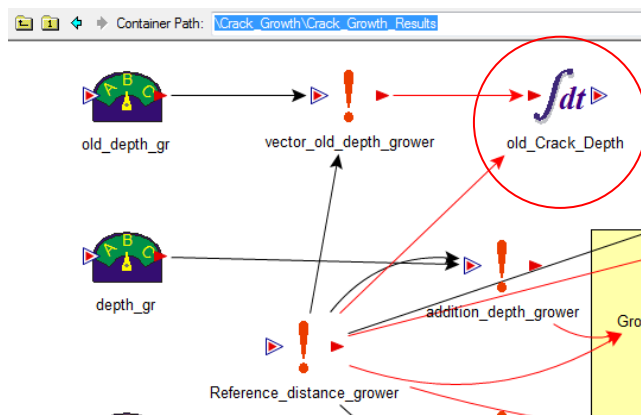
Define Triggering...

Define Triggering Events

Type	Trigger Definition
On Changed	~LoopCount

☐ For simultaneous events, act once

For the Integrator element “old_Crack_Depth,” the element is defined as a vector for the 19 cracks. The element should be driven by the Discrete Change elements “vector_old_depth_grower” and “Reference_distance_grower.” Therefore, these elements are listed in the “Discrete Change” field for this element (i.e., Discrete Change = **Reference_distance_grower;vector_old_depth_grower**).



Integrator Properties : old_Crack_Depth (Result Mode)

Definition Advanced

Element ID: Appearance...

Description:

Display Units: Type... Vector[nb_cracks]

Definition

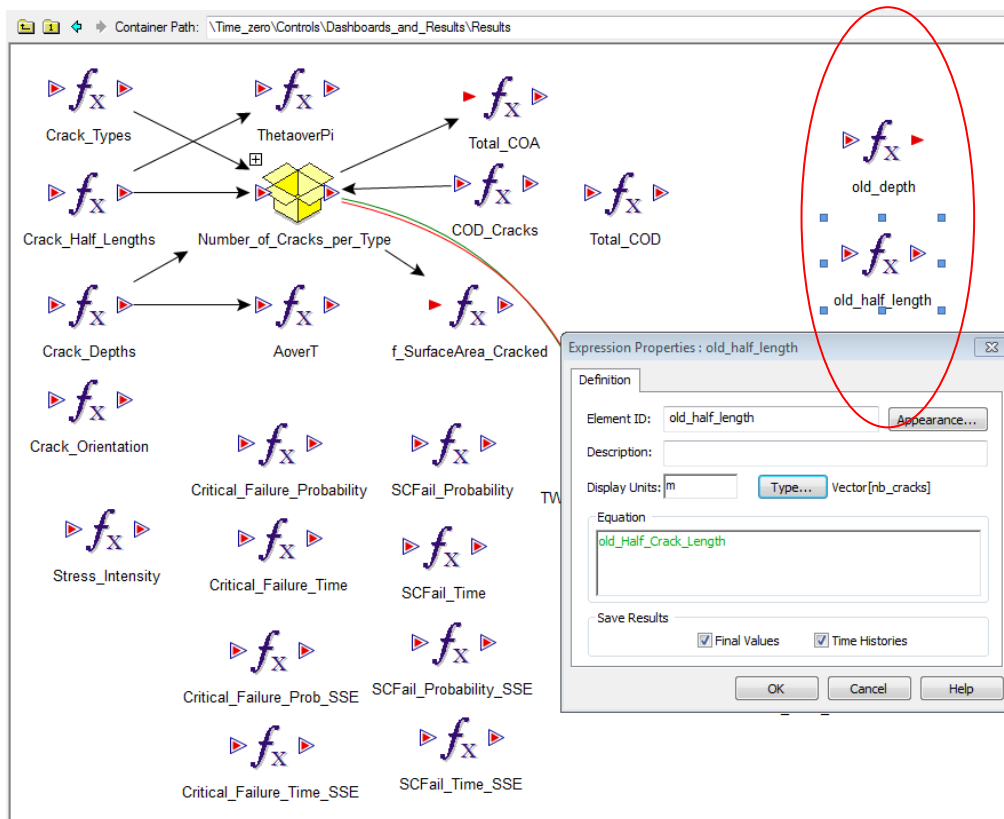
Initial Value:

Rate of Change:

Discrete Change:

Save Results ☐ Final Values ☐ Time Histories

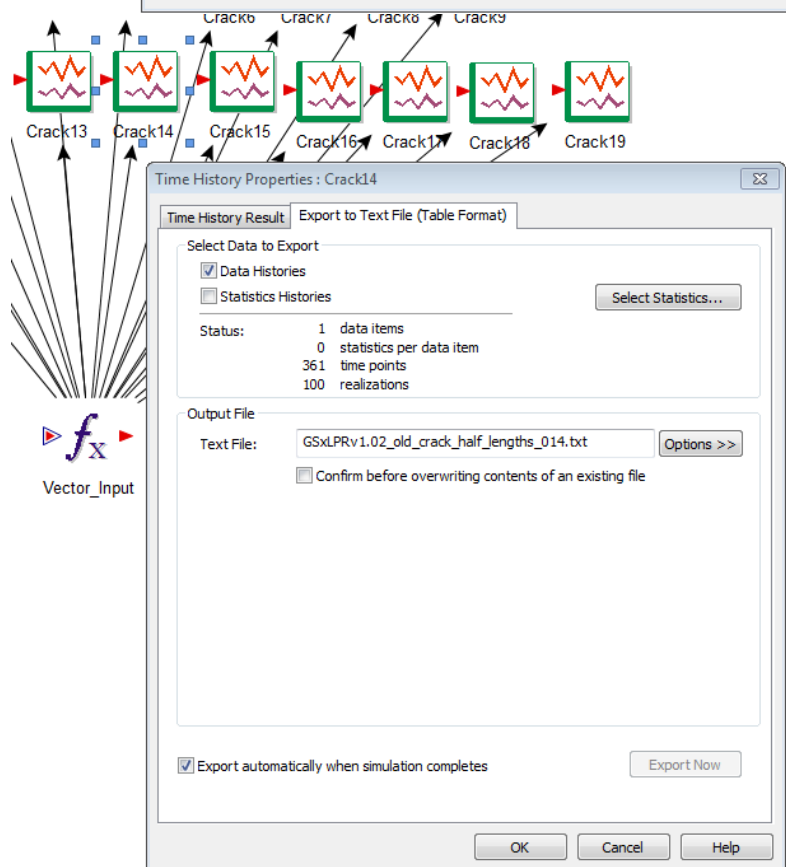
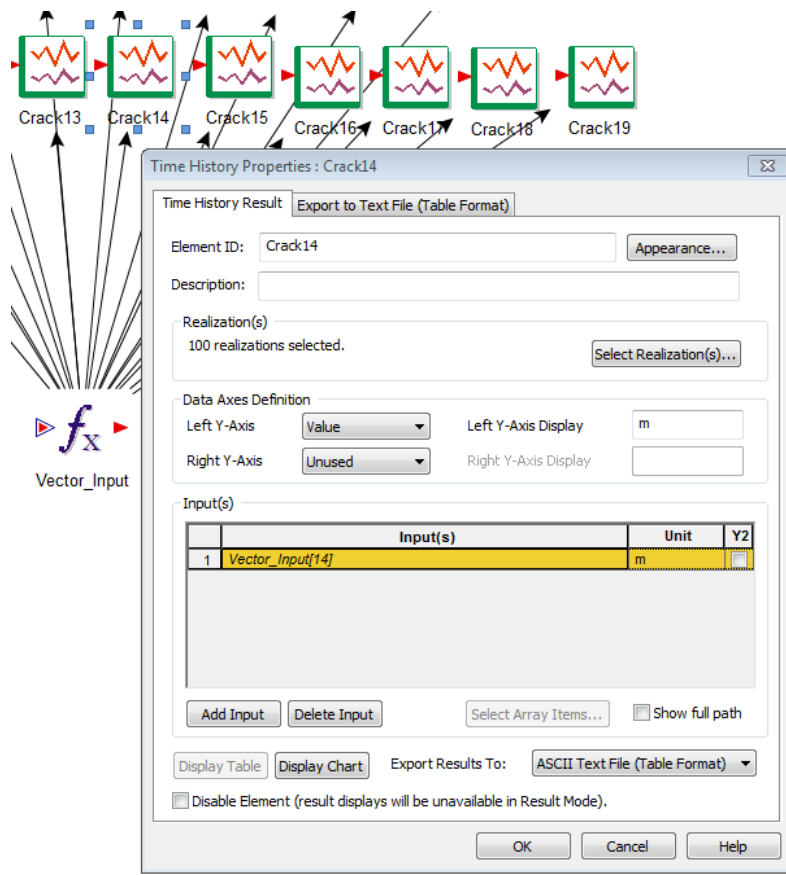
The vector expression (tracking the 19 cracks) represented by the Integrator Elements (“old_Crack_Depth” and “old_Half_Crack_Length”) is the vector calculated for the given timestep of a realization. For convenience, expression elements used in the results were collected in the \Time_zero\Controls\Dashboards_and_Results\Results container. Following this approach, expressions for the crack depth and half-lengths were added into this container and linked to the respective expression elements (“old_Crack_Depth” and “old_Half_Crack_Length”) contained in \Crack_Growth\Crack_Growth_Results as shown in the following.



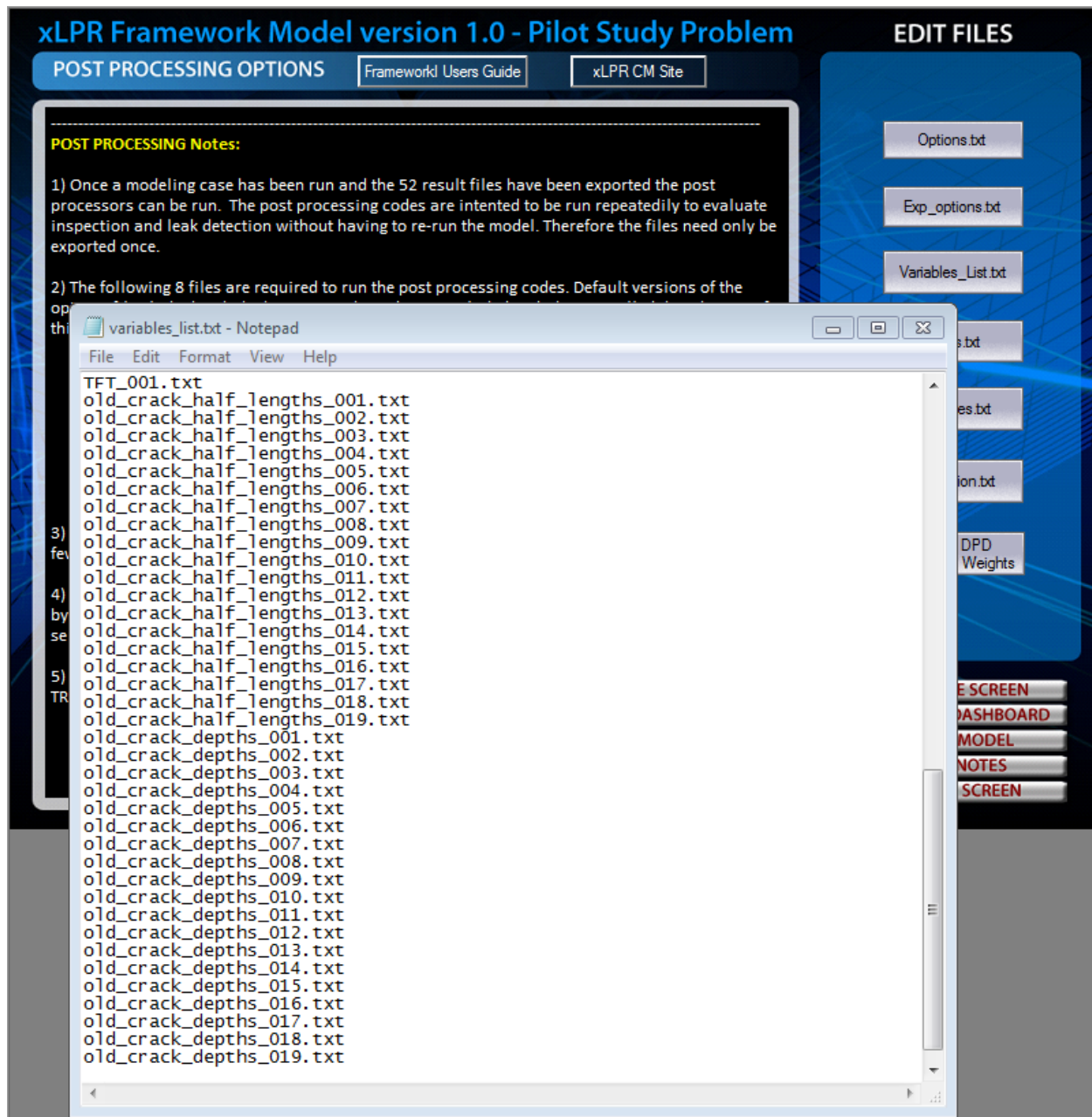
In regard to the elements contained in \Time_zero\Controls\Dashboards_and_Results\Vector_Export_Results, the Vector_Input values for each localized container (old_crack_half_lengths and old_crack_depths) should be linked to the respective values for crack half-length and depth collected in the \Crack_Growth\Crack_Growth_Results container as shown in the following screenshot for the old_half_length.



Each of the individual time history elements for cracks 1 to 19 for both containers (old_crack_half_lengths and old_crack_depths) was modified so that the appropriate vector element (representing the crack number) was tracked and a file name for the variable data export was input as shown in the following two screenshots.



Finally, the new exported parameters must be added to the Variables_List.txt input file for post-processing. As shown in the following screenshot, the file names for all of the exported variable values were added to the variables_list.txt document.



Implementing the CrackEnhancer Module Into the XLPR-SIAM Framework

The following details the steps taken to implement the module.

1. First, it was necessary to add the new crack enhancement parameters to the framework. To do this, the model implementation requires several substeps:


```

#=====
# Adding custom parameters here
#=====
# NOTE: Here adding definitions for crack enhancement
# parameters.
#-----
# Zcracklengthenhance
mean=999;stdv=9
args=[]
args.append( mean )
args.append( 'nondim' )
kwds = variateDao.createUniformVariate( 'Zcracklengthenhance', mean, stdv, args[1], 'epistemic' )
self.Zcracklengthenhance = xLPRVariate( *args, **kwds )
self.Zcracklengthenhance.setReqdUnits( 'nondim' )
self.Zcracklengthenhance.setUnitsType( 'nondim' )
self.Zcracklengthenhance.setDescription( 'A custom parameter to enhance crack length values' )
# cracksizeenhance
mean=888;stdv=8
args=[]
args.append( mean )
args.append( 'nondim' )
kwds = variateDao.createUniformVariate( 'Zcracksizeenhance', mean, stdv, args[1], 'epistemic' )
self.Zcracksizeenhance = xLPRVariate( *args, **kwds )
self.Zcracksizeenhance.setReqdUnits( 'nondim' )
self.Zcracksizeenhance.setUnitsType( 'nondim' )
self.Zcracksizeenhance.setDescription( 'A custom parameter to enhance crack size values' )

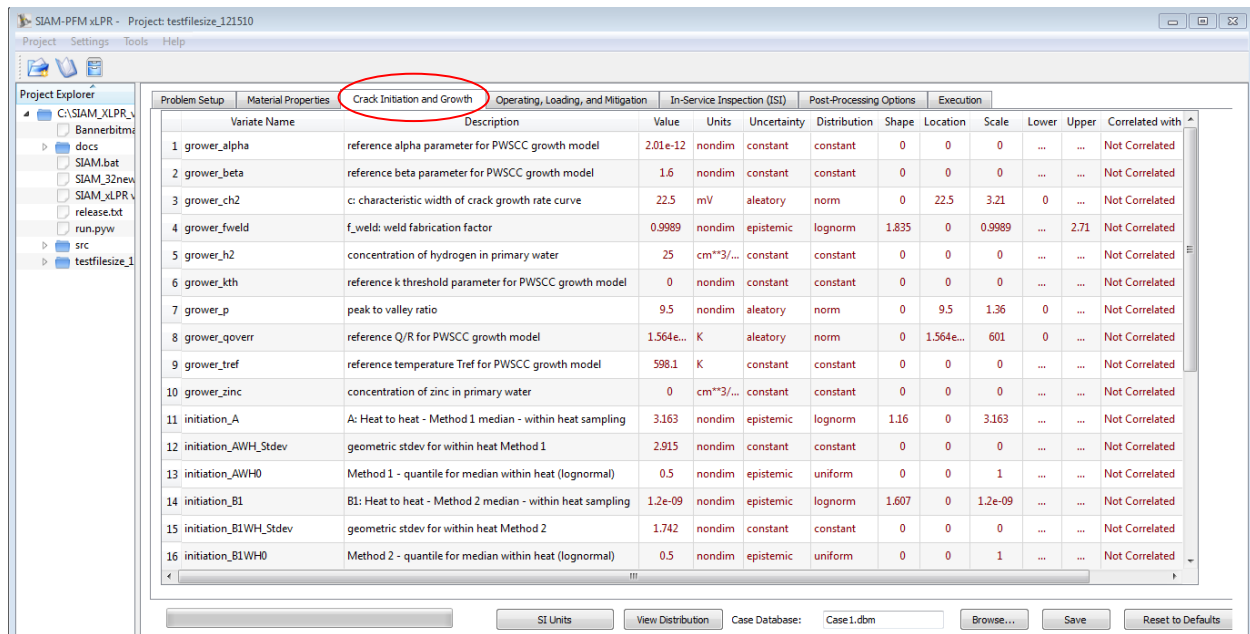
```

Zcracksizeenhance in the previous source code is the parameter used for enhancing the crack depth.

This step was particularly straightforward due to the predefined xLPRVariate class (part of the data structure developed for the XLPR-SIAM framework). Once added into the crack container, the variables of type xLPRVariate are initialized and are saved to the database project file when the program is run. However, to adjust the parameter values, the variables must be added to the display on the graphical user interface (GUI) to permit user interaction and adjustment of the parameter values.

- b. The next task required for adding the crack enhancement parameters was to add the crack enhancement parameters to be displayed on the GUI so that users can update/adjust parameter values. This modification requires some additional knowledge of PyQt and can potentially be complex depending on the nature of the changes needed. The XLPR-SIAM source code along with Summerfield (2007) was used to determine what adjustments would be needed to add the parameter interface to the GUI.

It was decided the easiest place to add the variable to the XLPR-SIAM GUI interface would be to the list of parameters provided in the “Crack Initiation and Growth” tab, as in the following screenshot:



To add the parameter to this tab, it is important to understand how the developers added variable values to the GUI display. When the XLPR-SIAM GUI is running, the parameter values are updated from the database project file and loaded into the Qt widgets for display (i.e., the tabbed lists of data). The parameter values displayed in the tabbed panes are generated by filtering lists of the variables contained in the database file.

The process is done in the following manner for the “Crack Initiation and Growth” tabbed pane parameter entries. The GUI is initiated when the SIAM_MainWindow.pyw python code is run (./src/view/SIAM_MainWindow.pyw). Upon loading or creating a project file, the main loop (the main loop for the Qt GUI) sets the file name from user input and then calls the xLPRModule() (a method defined in the SIAM_MainWindow.pyw file).

This method in turn calls the class object xLPRDIg(args.), which is the ./src/view/customviews/TableVariateViewModule.py class VariateTableView. This object contains methods that handle a number of operations (e.g., loading and saving variables, linking to post-processing) used by elements of the GUI contained in the tabbed panes (i.e., the tabbed panes in the previous screenshots). Most importantly for the current task, the class has a method for creating tabled lists of variables (i.e., tables of parameters): createView. This method contains parameters to specify options for the table of parameters object. The method uses an imported class VariateTableModel, which is defined in the file ./src/view/model/TableModelViewModule.py, to create the actual Qt table of parameters object (the Qt table widget). This class contains several methods that are used to order and display the data set contained in the table [e.g., loading data values from the database file, handling setting data entry values, parsing and creating lists of variables (epistemic, aleatory, constants), specifying the column entries (‘Value,’ ‘Uncertainty,’ ‘Distribution,’ ...), and sorting based the row order based upon selection of a particular column parameter.] The important part of this class is that a filter is applied to retrieve

variable values. In particular for the variables displayed on the “Crack Initiation and Growth” tabbed pane, the values are filtered using the filterFlag ‘initiation.’ This section of the source code (./src/view/model/TableModelViewModule.py) is near the comment entry (starting near line 623).

```
#=====
# Variate Class filters
#=====
if self.filterClass == 'variate':
    #=====
    # captures all variates
    #=====
    if self.filterFlag == 'all':
        self.variates = self.combinedVariateList
    #=====
    # captures crack initiation model variates
    #=====
    elif self.filterFlag == 'initiation':
        for var in self.combinedVariateList:
            name = var.getName()
            qname = QString( name )
            if qname.startsWith( QString('initiation'), Qt.CaseInsensitive ):
                self.variates.append( var )
            elif qname.startsWith( QString('grower_'), Qt.CaseInsensitive ):
                self.variates.append( var )
            #elif qname.startsWith( QString('cod_'), Qt.CaseInsensitive ):
            #self.variates.append( var )
            elif qname.startsWith( QString('Zcrack'), Qt.CaseInsensitive ):
                self.variates.append( var )
```

The additional “else if” statement is added to parse the variable name to check matching to the string pattern ‘Zcrack.’ If true, the variable is appended to the list of variables to be included in the table. This ensures that the newly added parameters Zcracksizeenhance and Zcracklengthenhance are included in the parameters displayed in the table for the “Crack Initiation and Growth” tabbed pane. Optionally, the use of a new filter is not explicitly required. For example, the parameter name can be selected such that it will be filtered to appear in the tab (i.e., “initiation_cracklengthenhance” could have been selected. In this case, there would be no need for an addition to the source code. A screenshot of the tabbed pane containing the added parameters follows.

Problem Setup		Material Properties	Crack Initiation and Growth	Operating, Loading, and Mitigation	In-Service Inspection (ISI)	Post-Processing Options	Execution					
	Variate Name	Description	Value	Units	Uncertainty	Distribution	Shape	Location	Scale	Lower	Upper	Correlated with
13	initiation_AWH0	Method 1 - quantile for median within heat (lognormal)	0.5	nondim	epistemic	uniform	0	0	1	Not Correlated
14	initiation_B1	B1: Heat to heat - Method 2 median - within heat sampling	1.2e-09	nondim	epistemic	lognorm	1.607	0	1.2e-09	Not Correlated
15	initiation_B1WH_Stdev	geometric stdev for within heat Method 2	1.742	nondim	constant	constant	0	0	0	Not Correlated
16	initiation_B1WH0	Method 2 - quantile for median within heat (lognormal)	0.5	nondim	epistemic	uniform	0	0	1	Not Correlated
17	initiation_C1	parameter for Method 3	0.04	nondim	constant	constant	0	0	0	Not Correlated
18	initiation_crack_dept...	initial crack depth	0.0015	m	aleatory	norm	0	0.0015	7.5e-05	0	...	Not Correlated
19	initiation_crack_half_L...	initial crack half-length	0.003	m	aleatory	norm	0	0.003	0.00015	0	...	Not Correlated
20	initiation_qverr	reference Q/R for PWSCC initiation module	2.2e+04	K	constant	constant	0	0	0	0	...	Not Correlated
21	initiation_RandU30	RandU3: Method 3 - initiation time sampling	0.5	nondim	aleatory	uniform	0	0	1	Not Correlated
22	initiation_RandULoc0	random number sampled for use in crack placement	0.5	nondim	aleatory	uniform	0	0	1	Not Correlated
23	initiation_SigTH	SigTH: threshold stress for Method 1	137.9	MPa	constant	constant	0	0	0	Not Correlated
24	initiation_subunits	number of circumferential subunits in weld	19	nondim	constant	constant	0	0	0	Not Correlated
25	initiation_XN1	parameter for Method 1	4	nondim	constant	constant	0	0	0	Not Correlated
26	initiation_XN3	parameter for Method 3	4	nondim	constant	constant	0	0	0	Not Correlated
27	Zcracklengthenhance	A custom parameter to enhance crack length values	999	nondim	epistemic	uniform	0	999	9	Not Correlated
28	Zcracksizeenhance	A custom parameter to enhance crack size values	888	nondim	epistemic	uniform	0	888	8	Not Correlated

From this interface, the values for items such as parameters and distribution type can be adjusted and saved to the database using the existing XLPR-SIAM framework structure.

2. The next step in implementing the crackenhancer module is to select the appropriate location to add an interface to the external FORTRAN module.
 - a. This was relatively straightforward as all FORTRAN modules use a common model interface through the TimeLoop module. The TimeLoop module is a FORTRAN code that is compiled with a FORTRAN to python (f2py) wrapper. The wrapper exposes selected subroutines and variables to python. In XLPR-SIAM, the parameter values for the TimeLoop module are set (passed) in the TimeLoopServiceModule (./src/service/TimeLoopServiceModule.py). In the next step, code was added to the TimeLoopServiceModule so that the values for crack enhancement parameters can be passed to the TimeLoop and, ultimately, a crack enhancement module.
 - b. For a given realization, all parameter values needed are loaded into the TimeLoop module (initialization block) and subsequent calls to the external modules follow the order and logic detailed in the xLPR Timeloop Flow Chart shown in Figure C-1, as excerpted from Klasky, et al. (2010b).

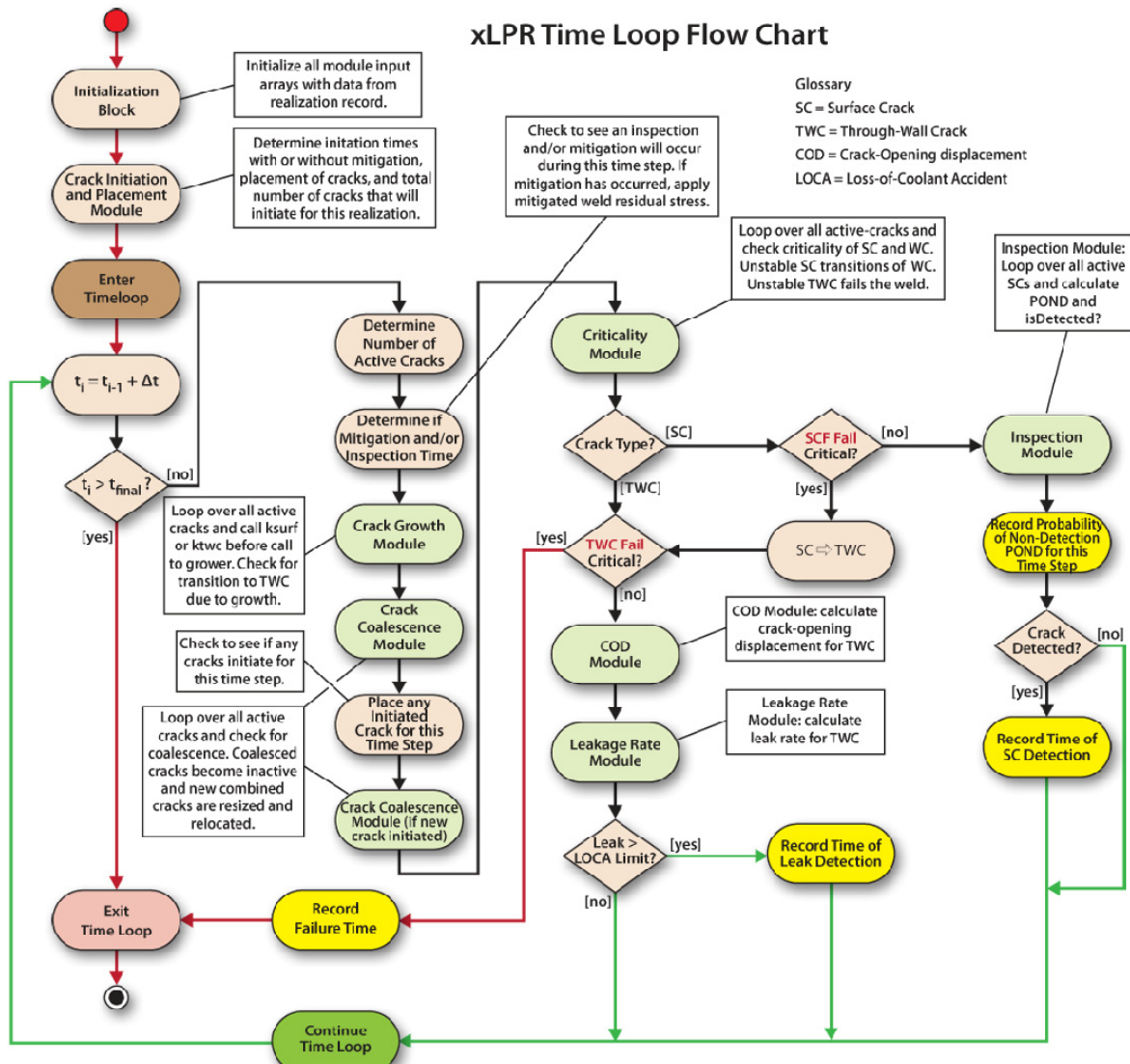


Figure C-1. The xLPR TimeLoop Execution Diagram Excerpted From Klasky, et al. (2010a)

- c. The crack enhancement module was added to the TimeLoop module sequence following the Crack Growth Module and before Crack Coalescence. The call to the crack enhancement module was added to the TimeLoop_V2.1.f90 source code.
3. The next step was to add the new crackenhancer module and interface for the parameters. This requires several substeps.
 - a. To pass the variables to the timeloop module, the crack enhancement parameter values must be passed to the timeloop module. The newly inserted variables in the timeloop module (TimeLoop_v2.1.f90) were named zcracklengthenhance and zcracksizenhance. At run time, the module is loaded as a timeloop object. The variable values are passed in the TimeLoopServiceModule. The following was added to the TimeLoopServiceModule code:

ADDED TO ./src/service/TimeLoopServiceModule.py:

```
#=====
# Load Zcracklengthenhance and Zcracksizenehance into timeloop module
timeloop.zcracklengthenhance = CC.Zcracklengthenhance.getValue()
timeloop.zcracksizenehance = CC.Zcracksizenehance.getValue()
```

- b. The parameters were added to the timeloop module source code. Input and output handling arrays were also defined for passing from the timeloop module and calling the crackenhancer module. Further, the call to the crack enhancer module is made. The following code was added to the TimeLoop_v2.1.f90 source code (notes are indicated to show the location in the source code where the additions were made).

ADDED TO TimeLoop_v2.1.f90:

Adding crack enhancement variables to the global namespace for the module timeloop. An additional output file is added and called for debugging and verifying parameter values.

```
!-----!
!       Adding variables for zcracklengthenhance and
!       zcracksizenehance
!-----!
real(8) :: zcracklengthenhance
real(8) :: zcracksizenehance
! adding a debugging filename (this should be commented out, just for
! testing verifying
CHARACTER(len=32) :: filename
```

*Adding debug output to TimeLoop_v2.1.f90 **subroutine write_timeloop_input_log** in order to verify passing of variables and maintain consistency with logging of variables.*

```
subroutine write_timeloop_input_log( f_log )
  implicit none
  character(LEN=*) , intent(in) :: f_log
  integer :: icrack, ierr, i
  real (8), external :: get_lognorm_q
  !-----!
  ! Compiler directive used to create the Python/C API bindings
  !-----!
  !f2py intent(in) f_log
  open(unit=10,file=f_log,status='UNKNOWN',position='APPEND', &
       action='WRITE', iostat=ierr)
  write(10,'(/' ' ***** ''')')
  write(10,'('' Input Data for Realization ',a6)' realization_key(1:6))
  write(10,'('' Realization Key = ',a20)' realization_key)
  write(10,'('' ***** ''')')

  write(10,'(/' ' ----- ''')')
  write(10,'('' MOD - Enhancement factors ''')')
  write(10,'('' ----- ''')')
  write(10,'('' zcracklengthenhance = ',ES12.5)' zcracklengthenhance)
  write(10,'('' zcracksizenehance = ',ES12.5)' zcracksizenehance)
```

*Adding input and output arrays for call from timeloop module to custom crackenhancer module in timeloop **subroutine run()**.*

```

!-----! Interface
variables for crackenhancer.v0.1.f90
!
real (8),dimension( 2 ) :: in_crackenhancer
real (8),dimension( 1 ) :: out_crackenhancer

      Initialize the input and output arrays in timeloop subroutine run().
!=====
! Initialize crackenhancer input/output interface arrays
!=====
in_crackenhancer(1) = zcracklengthenhance
in_crackenhancer(2) = zcracksizeenhance
in_crackenhancer(3) = 0
in_crackenhancer(4) = 0
out_crackenhancer(1) = 0
out_crackenhancer(2) = 0

```

Opening the additional output file for debugging and verifying parameter values.

```

!=====
! Opening debugging file
filename='crackenhancerdebug.txt'
open(unit=101,file=filename,status='UNKNOWN',position='APPEND',&
      action='WRITE',iostat=ierr)
write(101,'(//' ***** ' ')')
write(101,'(' Input Data for Realization ',a6)') realization_key(1:6)
write(101,'(' Realization Key = ',a20)') realization_key
write(101,'(' ***** ' ')')

write(101,'(//' ----- ' ')')
write(101,'(' MOD - Enhancement factors          ' ')')
write(101,'(' ----- ' ')')
write(101,'(' zcracklengthenhance = ',ES12.5)') zcracklengthenhance
write(101,'(' zcracksizeenhance = ',ES12.5)') zcracksizeenhance

```

The call to the crackenhancer module is made following grower. The call to the crack enhancement module is made only if the crack type is set to the value of -1 for a surface crack, consistent with the same call made in XLPR-GoldSim.

```

...
if (crack_ctype(icrack) .LT. 0) then
  call grower( method, state, in_grower, out_grower)
  time_grower = time_grower + out_grower(1)
  a           = out_grower(2)      ! crack depth [m]
  c           = out_grower(3)      ! crack half-length after grower[m]
  error_grower = int(out_grower(4))! error flag [-]
  ! -----
  ! Adding interface to crackenhancer module
  ! -----
  ! load current values into crackenhancer input vector
  in_crackenhancer(1) = zcracklengthenhance
  in_crackenhancer(2) = zcracksizeenhance
  in_crackenhancer(3) = a ! crack depth [m]
  in_crackenhancer(4) = c ! crack half-length after grower
  write(101,'(' Crack Initiated = ',i12)') icrack
  write(101,'(' a before      = ',ES12.5)') a
  write(101,'(' c begore     = ',ES12.5)') c
  if (crack_ctype(icrack) .EQ. -1 ) then
    call crackenhancer( in_crackenhancer, out_crackenhancer )
  endif
  ! update a and c values
  a = out_crackenhancer(1)
  c = out_crackenhancer(2)
  write(101,'(' a after      = ',ES12.5)') a
  write(101,'(' c after     = ',ES12.5)') c
  ! -----

```

```

! surface crack (SC) becomes a thru-wall crack (TWC) by growing
! through the wall
! -----
if ( a .GT. thickness ) then
    ...

```

- c. The source code for the crack enhancer module is similar to the source code used on the version for XLPR-GoldSim (the calculation is the same); the difference between the source code used for XLPR-GoldSim and XLPR-SIAM is the additional logic statements required for the XLPR-GoldSim version to interact with GoldSim. The source code for the XLPR-SIAM crackenhancer version follows (also see file crackenhancer_v0.1_XLPR-SIAM.f90):

```

! This is a simple add-on module to test the complexity of adding an
! additional module to SIAM_xLPR. The purpose of the module is to
! multiply crack length and crack depth values by the respective
! enhancement factors.
Subroutine crackenhancer(in, out)

    implicit none

    real(8), dimension(4) :: in
    real(8), dimension(2) :: out

    real(8) :: retavalue = 0
    real(8) :: retcvalue = 0

    real(8) :: cracklengthenhance = 0
    real(8) :: crackdepthenhance = 0

    real(8) :: crackdepth = 0 ! crack depth [m]
    real(8) :: crackhlflen = 0 ! crack half-length [m]

    real(8), dimension(2) :: outvector

    crackdepthenhance = in(2)
    cracklengthenhance = in(1)
    crackdepth = in(3)
    crackhlflen = in(4)

    retavalue = crackdepthenhance*crackdepth
    retcvalue = cracklengthenhance*crackhlflen

    out(1) = retavalue
    out(2) = retcvalue

end subroutine crackenhancer

```

4. The last step is compiling the implementation with the crack enhancement module. To do this, the source code for the timeloop module, along with the crackenhancer module, was compiled. For the interface to python, the timeloop module needs to be compiled as a PYD (python dynamic file) using f2py, the FORTRAN-to-python interface generator program. A wrapper file is needed to generate the pyd file with f2py. This can be generated by hand and/or automatically using f2py. The authors of XLPR-SIAM have added an additional FORTRAN-to-Python utility (f2py utility) to facilitate making the necessary calls to f2py.
 - a. The FORTRAN source code files were compiled as normal. This can be done from the command line or by using the f2py utility.

- b. The process for building a wrapper is detailed in Klasky, et al. (2010b) (see Chapter 4) and was followed exactly.
- c. When attempting to create the wrapper file (i.e., compiling the wrapper file), the process failed on the local machine. The following error output was observed in the captured output in the f2py GUI interface:

```
c:/gcc/bin/./lib/gcc/i686-pc-mingw32/4.5.1/././././i686-pc-mingw32/bin/ld.exe:
cannot find -lmsvcr71
collect2: ld returned 1 exit status
```

- d. The wrapper can be built manually from the command line using the f2py configuration file (pyf) (as generated from the f2py Utility). The lmsvcr71 flag may be the cause of the problems on the test system. Because the compilation command call (and flags) is hardwired into the f2py Utility and cannot be deselected, the compilation was done manually from the command line. To compile the PYD wrapper, the following command and flags were used:

```
c:\python26\python.exe c:\python26\scripts\f2py.py --
verbose --fcompiler=gnu95 --compiler=mingw32 -c
TimeLoopSignature.pyf Coalescence... <list of .f90 files> ...
.f90
```

This issue could be a problem with the local system setup. However, the issue for the local machine has not been explored further. Test runs of the f2py Utility on several other systems appear to work without this issue for recompilation of the original installation files.

The compiled PYD file is the ./src/service/temp directory. This is a temporary directory that is created by the f2py Utility. When the “Commit Wrapper to Service Layer” is selected, the compiled PYD file is copied from this directory and used to write over the TimeLoopWrapper.pyd file in the ./src/service/timeloop directory.

5. To fully assess the flexibility of the frameworks for exposing of parameter values for output and post-processing, an additional task was performed to expose the crack half-length and depth values before modification by the “new” crack enhancer module to external logging and post-processing. The process is detailed as follows.

In XLPR-SIAM, parameter data are exported for post-processing by opening hard-coded data streams to write to external text files during the program execution for all realizations. The initial data stream links to external text files are initiated in the RealizationsControllerModule where calls are made to the ./src/utlils/xLPRTTools.py method createWriteOnlyTextFile to create data streams to external files for appending parameter values at each timestep for all realizations. Note that the streams used for this purpose in XLPR-SIAM are Qt objects (i.e., Qt.QFile and Qt.QTextStream). When a realization is run through the timeloop module, arrays are defined to collect the parameter values for each timestep. When the realization completes, the values are passed back to python and then written to the file streams in the TimeLoopServiceModule at the end of each realization.

To add/expose new variables for post-processing, the source code for several files must be edited. Additionally, the timeloop module must be edited and recompiled as a PYD file. The steps required to expose the crack length and crack depth values (before modification by the crackenhancer module) are detailed next.

Data dictionaries are used to pass and recall file stream objects, are created in ./src/controller/RealizationsControllerModule.py, and are also called/used to update values for each realization in ./src/service/TimeLoopServiceModule.py. First, the file/stream objects must be created.

When the TimeLoopServiceModule method executeTimeLoop is called (running a realization), the stream dictionaries are passed to the method. To implement these changes, the following code was added to the ./src/controller/RealizationsControllerModule.py file (note the edits are excerpted from the file with several lines of code removed; newly added code has been highlighted).

< Here, a number of lines of code are omitted. >

```
def createOutputDataDict(self, numE, numA, pipeWeld, loadContainer):
    """
    create dictionaries that contain arrays of output data for post-processing
    """
    #=====
    # determine array sizes
    #=====
    self.numE      = numE
    self.numA      = numA
    self.tfinal    = loadContainer.tfinal.getValue()
    self.time_step = loadContainer.time_step.getValue()
    numTimeSteps   = int(self.tfinal/self.time_step)
    self.numTimeSteps = numTimeSteps + 1

    #=====
    # load into dictionary
    #=====
    self.output_data_dict['depth']      = 3
    self.output_data_dict['half_length'] = 3
    self.output_data_dict['old_depth']  = 3
    self.output_data_dict['old_half_length'] = 3
    self.output_data_dict['surfk0']     = 3
    self.output_data_dict['surfk90']    = 3
    self.output_data_dict['pnd']        = 3
    self.output_data_dict['leakrate']    = 3
    self.output_data_dict['coa']         = 3

    self.output_data_dict['FracArea']    = 1
    self.output_data_dict['total_leakrate'] = 1
    #self.output_data_dict['SC_time']     = 1
    self.output_data_dict['FirstLeak']    = 1
    self.output_data_dict['Rupture']      = 1
    self.output_data_dict['Rupture_SSE'] = 1
    self.output_data_dict['FirstInit']    = 1
    self.output_data_dict['COA1']         = 1
    self.output_data_dict['COA3']         = 1
```

< Here, a number of lines of code are omitted. >

< In method def createExecuteRealization >

```
#=====
# create dictionaries to hold IO QTextStreams
# provide for first 3 initiating cracks
#=====
```

```

self.stream_dict_depth = {}
self.stream_dict_old_depth = {} # Custom tracking for non-modified crack depth
self.stream_dict_half_length = {}
self.stream_dict_old_half_length = {} # Custom tracking for non-modified half_length
self.stream_dict_surfk0 = {}
self.stream_dict_surfk90 = {}
self.stream_dict_coa = {}
self.stream_dict_leakrate = {}

for i in range(19):
    icrk = "%03d" % (i+1)
    pnd_fname = "pnd_" + icrk + ".txt"
    ctype_fname = "ctype_" + icrk + ".txt"
    ( stream_pnd, fh_pnd ) = Tools.createWriteOnlyTextFile( pnd_fname,
                                                            self.postPath, prefix )
    ( stream_ctype, fh_ctype ) = Tools.createWriteOnlyTextFile( ctype_fname,
                                                                self.postPath, prefix )
    self.stream_dict_pnd[ pnd_fname ] = ( stream_pnd, fh_pnd, pnd_fname )
    self.stream_dict_ctype[ ctype_fname ] = ( stream_ctype, fh_ctype, ctype_fname )

for i in range(3):
    icrk = "%03d" % (i+1)
    depth_fname = "depth_" + icrk + ".txt"
    old_depth_fname = "old_depth_" + icrk + ".txt"
    half_length_fname = "half_length_" + icrk + ".txt"
    old_half_length_fname = "old_half_length_" + icrk + ".txt"
    surfk0_fname = "surfk0_" + icrk + ".txt"
    surfk90_fname = "surfk90_" + icrk + ".txt"
    coa_fname = "coa_" + icrk + ".txt"
    leakrate_fname = "leakrate_" + icrk + ".txt"

    ( stream_depth, fh_depth ) = Tools.createWriteOnlyTextFile( depth_fname,
                                                                self.postPath, prefix )
    ( stream_old_depth, fh_old_depth ) = Tools.createWriteOnlyTextFile(
old_depth_fname, self.postPath, prefix )
    ( stream_half_length, fh_half_length ) = Tools.createWriteOnlyTextFile(
half_length_fname, self.postPath, prefix )
    ( stream_old_half_length, fh_old_half_length ) = Tools.createWriteOnlyTextFile(
old_half_length_fname, self.postPath, prefix )
    ( stream_surfk0, fh_surfk0 ) = Tools.createWriteOnlyTextFile( surfk0_fname,
                                                                self.postPath, prefix )
    ( stream_surfk90, fh_surfk90 ) = Tools.createWriteOnlyTextFile( surfk90_fname,
                                                                self.postPath, prefix )
    ( stream_coa, fh_coa ) = Tools.createWriteOnlyTextFile( coa_fname,
                                                            self.postPath, prefix )
    ( stream_leakrate, fh_leakrate ) = Tools.createWriteOnlyTextFile(
leakrate_fname, self.postPath, prefix )

    #print("i = ",i)
    #print("stream_old_depth: ",stream_old_depth)

    self.stream_dict_depth[ depth_fname ] = ( stream_depth, fh_depth,
                                                depth_fname )
    self.stream_dict_old_depth[ old_depth_fname ] = ( stream_old_depth,
                                                        fh_old_depth,
                                                        old_depth_fname )
    self.stream_dict_half_length[ half_length_fname ] = ( stream_half_length,
                                                            fh_half_length,
                                                            half_length_fname )
    self.stream_dict_old_half_length[ old_half_length_fname ] = ( stream_old_half_length,
                                                                    fh_old_half_length,
                                                                    old_half_length_fname )
    self.stream_dict_surfk0[ surfk0_fname ] = ( stream_surfk0, fh_surfk0,
                                                surfk0_fname )
    self.stream_dict_surfk90[ surfk90_fname ] = ( stream_surfk90, fh_surfk90,
                                                surfk90_fname )
    self.stream_dict_coa[ coa_fname ] = ( stream_coa, fh_coa, coa_fname )
    self.stream_dict_leakrate[ leakrate_fname ] = ( stream_leakrate,
                                                    fh_leakrate,
                                                    leakrate_fname )

```


< Here, a number of lines of code are omitted. >

```
#=====
# carry out timeloop execution for this trial
#=====
TimeLoopService.executeTimeLoop( trialObject, self.debug, self.deterministic,
                                self.stream_dict,
                                self.stream_dict_pnd,
                                self.stream_dict_ctype,
                                self.stream_dict_other,
                                self.stream_dict_depth,
                                self.stream_dict_old_depth,
                                self.stream_dict_half_length,
                                self.stream_dict_old_half_length,
                                self.stream_dict_surfk0,
                                self.stream_dict_surfk90,
                                self.stream_dict_coa,
                                self.stream_dict_leakrate,
                                base )
```

Now that the stream/file objects are created and are passed to the TimeLoopService module for each execution of a realization, the next step is to edit the TimeLoop module and ensure that the crack length and depth values are captured into arrays. The ./src/service/timeloop/TimeLoop_V2.1_MOD.f90 source code is edited as follows. First, new parameters were defined (olddepth and oldhalf) to pass the crack length and depth values before modification by the crackenhancer module. Arrays of dimension 720 × 19 are created to hold the parameter values for each timestep and crack id, respectively. Values for the crack length and depth are captured during looping over cracks and timesteps in the parameter olddepth and oldhalf. These values are then written into the appropriate elements in the crack_old_depth and crack_old_half_length and ultimately passed to the appropriate elements in the arrays crack_old_depth_log and crack_old_half_length_log, which are populated with the values for all timesteps and cracks. The values contained for these arrays are exposed to the python TimeLoopServiceModule.

< Here, a number of lines of code are omitted. >

```
!-----!
!       Adding variables for zcracklengthenhance and
!       zcracksizeenhance
!-----!
real(8) :: zcracklengthenhance
real(8) :: zcracksizeenhance
real(8) :: olddepth
real(8) :: oldhalf
! adding a debugging filename (this should be commented out, just for
! testing verifying
CHARACTER(len=32) :: filename
```

< Here, a number of lines of code are omitted. >

```
!-----!
! output data for logging and post-processing of 1st three cracks
!-----!
integer,dimension(0:720,19) :: crack_ctype_log           ! [-]
real(8),dimension(0:720,19) :: crack_depth_log           ! [m]
real(8),dimension(0:720,19) :: crack_old_depth_log       ! [m]
real(8),dimension(0:720,19) :: crack_half_length_log     ! [m]
real(8),dimension(0:720,19) :: crack_old_half_length_log ! [m]
real(8),dimension(0:720,19) :: crack_surfk0_log          ! [MPa-m**0.5]
real(8),dimension(0:720,19) :: crack_surfk90_log         ! [MPa-m**0.5]
real(8),dimension(0:720,19) :: crack_average_coa_log     ! [m**2]
real(8),dimension(0:720,19) :: crack_leakrate_log        ! [m**3/sec]
```

```

real(8),dimension(0:720,19) :: crack_pond_log           ! [-]
real(8),dimension(0:720,19) :: crack_frac_area_log      ! [m**2]

```

< Here, a number of lines of code are omitted. >

```

!-----!
!  variable data for cracks
!-----!
real(8),dimension(30) :: crack_depth           ! [m]
real(8),dimension(30) :: crack_old_depth       ! [m]
real(8),dimension(30) :: crack_half_length     ! [m]
real(8),dimension(30) :: crack_old_half_length ! [m]
real(8),dimension(30) :: crack_initiationtime  ! [yr]
real(8),dimension(30) :: crack_leakrate        ! [m**3/s]
real(8),dimension(30) :: crack_coa             ! [m**2]
real(8),dimension(30) :: crack_coa_1          ! [m**2]
real(8),dimension(30) :: crack_coa_3          ! [m**2]
real(8),dimension(30) :: crack_location        ! [rad]
real(8),dimension(30) :: crack_timeoffailure   ! [yr]
real(8),dimension(30) :: crack_timeoffailure_sse ! [yr]
real(8),dimension(30) :: crack_timeofdetection ! [yr]
real(8),dimension(30) :: crack_timeofleakdetection ! [yr]
real(8),dimension(30) :: crack_blwh            ! [-]

```

< Here, a number of lines of code are omitted. >

< Now, in subroutine run() >

```

!=====
!  initialize all logging variables
!=====
isFirstLeak      = .TRUE.
isRupture_SSE    = .TRUE.
isFirstInitiation = .TRUE.
isCOA_1          = .TRUE.
isCOA_3          = .TRUE.
crack_ctype_log  = 0
crack_depth_log  = zero
crack_old_depth_log = zero !
crack_half_length_log = zero
crack_old_half_length_log = zero !
crack_leakrate_log = zero
crack_surfk0_log  = zero
crack_average_coa_log = zero
crack_pond_log    = zero
crack_surfk90_log = zero
total_leakrate_log = zero
first_leak_log    = zero
coa_1_log         = zero
coa_3_log         = zero
rupture_log       = zero
rupture_sse_log   = zero
first_initiation_log = zero

```

< Here, a number of lines of code are omitted. >

```

!=====
!  initialization section for local namespace
!=====
pi          = 4.0d0*atan(one) ! pi to machine precision [-]
average_cod = zero           ! crack-opening-displacement total [m]
current_time = zero          ! [yr]
previous_time = -time_step    ! [yr]
!
tref        = grower_tref      !pwscc parameter eqn 13 of program plan [K]
qoverr      = grower_qoverr    !pwscc parameter eqn 13 of program plan [K]
alpha       = grower_alpha     !pwscc parameter eqn 13 of program plan [-]
beta        = grower_beta      !pwscc parameter eqn 13 of program plan [-]

```

```

kth          = grower_kth          !pwscc parameter eqn 13 of program plan    [MPa (m)**0.5]
!
temp         = temperature          !temperature                          [C]
tinterval    = time_step            !time interval                      [yr]
thickness    = pipe_wall_thickness  !pipe wall thickness              [m]
rinner       = 0.5d0*pipe_inner_diameter !inner pipe radius                [m]
router       = 0.5d0*pipe_outer_diameter !outer pipe radius                [m]
rmean        = 0.5d0*(rinner+router) !mean radius                      [m]
rovert       = rinner/thickness      !inner radius normalized by the wall thickness [-]
sigflow      = 0.5d0*(weld_yieldstrength+weld_ultimatestrength) ! material flow stress[MPa]
axialload    = dw_fx + te_fx + ts_fx ! total axial load without pressure [kN]

next_inspection = inspection_interval ! initialize inspection time
crack_isdetected = .false.
crack_isleakdetected = .false.
crack_timeoffailure = 61.0d0
crack_timeoffailure_sse = 61.0d0
crack_timeofdetection = zero
crack_timeofleakdetection = zero
crack_depth = zero ! [m]
crack_old_depth = zero ! [m]
crack_half_length = zero ! [m]
crack_old_half_length = zero ! [m]
crack_leakrate = zero ! [m**3/s]
crack_coa = zero ! [m**2]
crack_coa_1 = zero ! [m**2] coa=1 inch equivalent break diameter
crack_coa_3 = zero ! [m**2] coa=3 inch equivalent break diameter

```

< Here, a number of lines of code are omitted. >

```

! -----
! MOD Adding interface to crackenhancer module
! -----
! load current values into crackenhancer input vector
in_crackenhancer(1) = zcracklengthenhance
in_crackenhancer(2) = zcracksizeenhance
in_crackenhancer(3) = a ! crack depth [m]
in_crackenhancer(4) = c ! crack half-length after grower
olddepth = a ! the initial value before modification
oldhalf = c ! the initial value before modification
crack_old_depth(icrack) = a ! track old crack depth [m]
crack_old_half_length(icrack) = c ! track old crack half length [m]
write(101, '(' a Crack Initiated = ',i12)') icrack
write(101, '(' a before = ',ES12.5)') a
write(101, '(' c begore = ',ES12.5)') c
if (crack_ctype(icrack) .EQ. -1 ) then
    call crackenhancer( in_crackenhancer, out_crackenhancer )
endif
! update a and c values
a = out_crackenhancer(1)
c = out_crackenhancer(2)
write(101, '(' a after = ',ES12.5)') a
write(101, '(' c after = ',ES12.5)') c

```

< Here, a number of lines of code are omitted. >

```

! -----
! load in logging data
! -----
if (icrack .LE. 19 ) then
    if (itime .EQ. 0) then
        crack_depth_log(itime,icrack) = crack_depth(icrack) ! current
        ! crack depth [m]
        crack_old_depth_log(itime,icrack) = crack_old_depth(icrack)
        crack_half_length_log(itime,icrack) = crack_half_length(icrack) ! current
        ! crack half length [m]
        crack_old_half_length_log(itime,icrack) = crack_old_half_length(itime,icrack)
    end if
end if

```

```

        crack_ctype_log(itime,icrack)      = crack_ctype(icrack)
        crack_surfk90_log(itime,icrack)    = max(surfk90,0.0d0)
        crack_surfk0_log(itime,icrack)     = max(surfk0,0.0d0)
        crack_frac_area_log(itime,icrack)  = frac_area
    else
        crack_depth_log(itime-1,icrack)    = crack_depth(icrack)      ! current
                                           !crack depth [m]
        crack_half_length_log(itime-1,icrack) = crack_half_length(icrack) ! current
                                           ! crack half length [m]

        crack_ctype_log(itime-1,icrack)    = crack_ctype(icrack)
        crack_surfk90_log(itime-1,icrack)  = max(surfk90,0.0d0)
        crack_surfk0_log(itime-1,icrack)   = max(surfk0,0.0d0)
        crack_frac_area_log(itime-1,icrack) = frac_area
    endif
endif
endif
! -----

! -----
! put a limit on the total crack length = 0.995 of inner circumference
! -----
if ( c .ge. 0.4975d0*pi*pipe_inner_diameter ) then
    crack_timeoffailure(icrack) = previous_time
    ! -----
    ! complete log entries for remaining timesteps
    ! -----
    if (itime .EQ. 0) then
        crack_ctype_log(itime:720,1:19)      = 200
        crack_depth_log(itime:720,icrack)    = crack_depth(icrack)
        crack_old_depth_log(itime:720,icrack) =
            crack_old_depth(icrack)
        crack_half_length_log(itime:720,icrack) = crack_half_length(icrack)
        crack_old_half_length_log(itime:720,icrack) =
            crack_old_half_length(icrack)
        crack_surfk90_log(itime:720,icrack)    = max(surfk90,0.0d0)
        crack_surfk0_log(itime:720,icrack)     = max(surfk0,0.0d0)
        crack_frac_area_log(itime:720,icrack)  = frac_area
        rupture_log(itime:720)                 = one
    else
        crack_ctype_log(itime-1:720,1:19)      = 200
        crack_depth_log(itime-1:720,icrack)    = crack_depth(icrack)
        crack_old_depth_log(itime-1:720,icrack) =
            crack_old_depth(icrack)
        crack_half_length_log(itime-1:720,icrack) = crack_half_length(icrack)
        crack_old_half_length_log(itime-1:720,icrack) =
            crack_old_half_length(icrack)
        crack_surfk90_log(itime-1:720,icrack)    = max(surfk90,0.0d0)
        crack_surfk0_log(itime-1:720,icrack)     = max(surfk0,0.0d0)
    endif
endif

```

At this point, the timeloop wrapper and timeloop PYD had to be recompiled. This was done using the same method previously described.

The crack_old_depth_log and crack_old_half_length_log arrays should be exposed to the TimeLoopService module. The next step is to modify the ./src/service/TimeLoopServiceModule.py and write the array values to the appropriate stream objects (export the data). To do this, the following edits were made to the ./src/service/TimeLoopServiceModule.py file. First, the class docs were updated to reflect the input of the stream/file objects, which are passed from the RealizationsControllerModule. Next, after the TimeLoop module has been called, the remainder of the TimeLoopServiceModule code passes through several loops, which facilitate writing of logging data and writing to the post-processing files. An important point to note here is that the operator "<<" is used in this section of the code behaves differently for the Qt stream objects than for python. Here, the "<<" adds the text characters to the Qt text stream (in turn appending the line of text to the open file).

```

class TimeLoopService(object):
    '''
    classdocs
    '''
    @staticmethod
    def executeTimeLoop( current_trial, debug, deterministic,
                        stream_dict,
                        stream_dict_pnd,
                        stream_dict_ctype,
                        stream_dict_other,
                        stream_dict_depth,
                        stream_dict_old_depth,
                        stream_dict_half_length,
                        stream_dict_old_half_length,
                        stream_dict_surfk0,
                        stream_dict_surfk90,
                        stream_dict_coa,
                        stream_dict_leakrate,
                        caseName ):
        '''
        Execute the TimeLoop for each realization
        write output to file and store in output datastore_out.tar.gz
        '''

```

< Here, a number of lines of code are omitted. >

```

for icrack in xrange(3):
    icrk = "%03d" % (icrack+1)

    depth_fname = "depth_" + icrk + ".txt"
    old_depth_fname = "old_depth_" + icrk + ".txt"
    half_length_fname = "half_length_" + icrk + ".txt"
    old_half_length_fname = "old_half_length_" + icrk + ".txt"
    surfk0_fname = "surfk0_" + icrk + ".txt"
    surfk90_fname = "surfk90_" + icrk + ".txt"
    coa_fname = "coa_" + icrk + ".txt"
    leakrate_fname = "leakrate_" + icrk + ".txt"

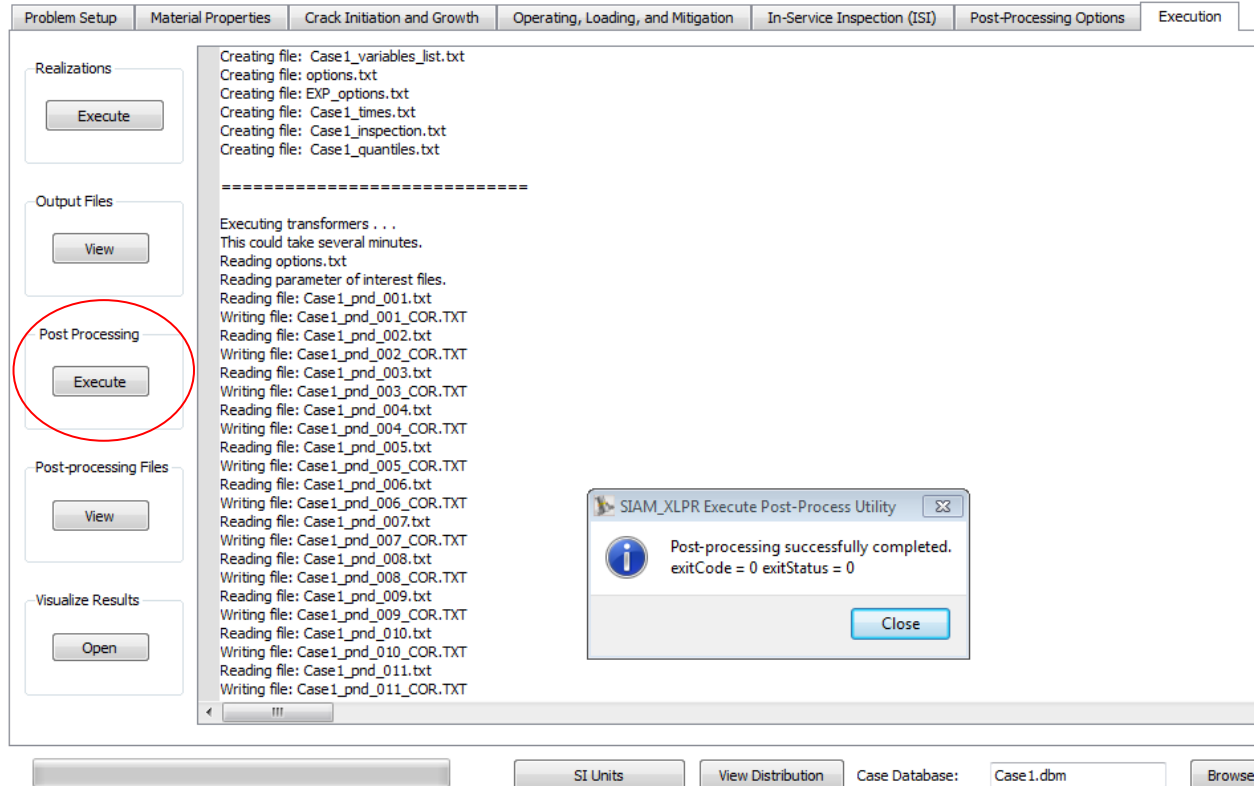
    (stream_depth, fh_depth, fname_depth) = stream_dict_depth[depth_fname]
    (stream_half_length, fh_half_length, fname_half_length) =
        stream_dict_half_length[half_length_fname]
    (stream_old_depth, fh_old_depth, fname_old_depth) =
        stream_dict_old_depth[old_depth_fname]
    (stream_old_half_length, fh_old_half_length, fname_old_half_length) =
        stream_dict_old_half_length[old_half_length_fname]
    (stream_surfk0, fh_surfk0, fname_surfk0) = stream_dict_surfk0[surfk0_fname]
    (stream_surfk90, fh_surfk90, fname_surfk90) =
        stream_dict_surfk90[surfk90_fname]
    (stream_coa, fh_coa, fname_coa) = stream_dict_coa[coa_fname]
    (stream_leakrate, fh_leakrate, fname_leakrate) = stream_dict_leakrate[leakrate_fname]

    for itime in xrange(num_time_steps):
        stream_depth << timeloop.crack_depth_log[itime,icrack] << tab
        stream_half_length << timeloop.crack_half_length_log[itime,icrack] << tab
        stream_old_depth << timeloop.crack_old_depth_log[itime,icrack] << tab
        stream_old_half_length << timeloop.crack_old_half_length_log[itime,icrack] << tab
        stream_surfk0 << timeloop.crack_surfk0_log[itime,icrack] << tab
        stream_surfk90 << timeloop.crack_surfk90_log[itime,icrack] << tab
        stream_coa << timeloop.crack_average_coa_log[itime,icrack] << tab
        stream_leakrate << timeloop.crack_leakrate_log[itime,icrack] << tab

    stream_depth << nl
    stream_half_length << nl
    stream_old_depth << nl
    stream_old_half_length << nl
    stream_surfk0 << nl
    stream_surfk90 << nl
    stream_coa << nl
    stream_leakrate << nl

```

Before, the post-processing option can be requested from the XLPR-SIAM GUI, an additional linkage must be created to ensure that the proper input is sent to the TRANSFORMERS and EXPECTATION programs (i.e., see the following screenshot):



When the post-processing button is selected, the `createPostProcessing` method of `./src/view/customviews` is called. The XLPR-SIAM GUI allows for user input of parameters for TRANSFORMERS and EXPECTATION, which are used to automatically generate the required input files for these programs. In adding this feature, the authors removed the direct user interface with the input text files for these programs and have recompiled TRANSFORMERS and EXPECTATION programs into one executable file. Because of this modification, the additional variables must be added to the source code for the `createPostProcessing` method of `./src/view/customviews.py`. This was done, and the source code edits follow.

< Here, a number of lines of code are omitted. >

```
def createPostProcessing(self):

    self.save()

    ( stream_options, fh_options )      = Tools.createWriteOnlyTextFile( 'options.txt',
                                                                           where=self.postPath )
    ( stream_exp_options, fh_exp_options ) = Tools.createWriteOnlyTextFile(
        'EXP_options.txt', where=self.postPath )
    baseName                             = unicode(QFileInfo( self.filename ).baseName())
    inspection_times                      = []

    try:
        #=====
        # create the "variables_list.txt" file
```

#=====

< Here, a number of lines of code are omitted. >

```
stream_var_list << "_old_depth_001.txt" << nl
stream_var_list << "_old_depth_002.txt" << nl
stream_var_list << "_old_depth_003.txt" << nl
```

```
stream_var_list << "_old_half_length_001.txt" << nl
stream_var_list << "_old_half_length_002.txt" << nl
stream_var_list << "_old_half_length_003.txt" << nl
```

References

GoldSim Technology Group LLC. "GoldSim User's Guide." Vols 1 and 2. Version 10.11. Issaquah, Washington: GoldSim Technology Group LLC. 2011.

Klasky, H.B., P.T. Williams, S. Yin, and B.R. Bass. "SIAM-xLPR Version 1.0 Framework Report." ORNL/NRC/LTR-248. Oak Ridge, Tennessee: Oak Ridge National Laboratory. 2010a.

Klasky, H.B., P.T. Williams, B.R. Bass, and S. Yin. "Structural Integrity Assessments Modular-Probabilistic Fracture Mechanics (SIAM-PFM): User's Guide for xLPR." ORNL/NRC/LTR-247. Oak Ridge, Tennessee: Oak Ridge National Laboratory. 2010b.

Summerfield, M. "Rapid GUI Programming With Python and QT." ISBN-10: 0132354187. Upper Saddle River, New Jersey: Prentice Hall. 2007.